Selective Enumeration
A Formal Definition

Craig A. Damon

January, 1998
CMU-CS-98-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored by the National Science Foundation (NSF) under Grant No. CCR-9523972, the Defense Advanced Research Projects Agency (DARPA) under Contract No. F33615-93-1330, and the National Institute of Standards and Technology (NIST) under Contract No. 60NANB7D0062.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of any of the research sponsors or of the United States Government.

Keywords

Relational calculus, exhaustive search, model checking, specification checking, constraint satisfaction.

Abstract

Selective enumeration is a method for reducing the number of cases required when performing a generate-and-test search to solve relational formulae. This paper gives a formal definition of selective enumeration and using that definition, proves soundness for each of the selective enumeration techniques developed.

1. Introduction

Sets, functions, and binary relations combine to provide a convenient, yet rigorous, framework for modeling software systems. Z [Spi92], probably the most widely used formal notation for describing software systems, is based entirely on these constructs. As sets, functions, and relations can all be described using relational formulae, I use the term *relational specification* to describe any specification built on these constructs.

Other software description notations also draw much of their expressive power from these constructs. Within the database community, the inter-relationships in a database schema are often specified using an entity-relationship diagram [Che76]. Given the name, it should not be surprising that entity-relationship diagrams can be clearly and succinctly described using relations.

More recently, UML [BJR97] has gathered great interest in the object community. Although UML combines several different notations to describe a single object design, many of these notations are built from sets, functions, and binary relations.

Despite the broad appeal of these constructs, little automated support is available for analyzing relational specifications. Theorem provers [ES94; SM96] can help, but they require enormous manual effort and provide little guidance to help repair faulty specifications. Model checkers [BC+92; CPS93] can analyze system specifications based on other formalisms, but no model checkers are available for relational specifications.

1.1 Generate-and-Test Searching

A method for solving relational formulae must lie at the core of any automated tool for analyzing relational specifications. The simplest approach is a generate-and-test search. A generate-and-test search generates every possible mapping of variables to values, called *assignments*, for a particular formula. The search then tests each generated assignment against that formula. The result of the search is a set of *satisfying assignments*, that is, assignments that give a true interpretation to the formula. A depth first search can trivially generate a complete set of assignments, with each level of the search tree corresponding to a distinct variable in the formula. Testing a single assignment against a formula is also straightforward, requiring only an implementation of the standard boolean, set and relational operations, making a generate-and-test search a simple solution.

However, using generate-and-test search as a solver presents two limitations. By its nature, a generate-and-test search will consider only some finite subset of the (generally infinite) possible assignment space. Although this limitation prevents a generate-and-test search from being a true verifier for infinite problems, it does not remove all practical applications. As I believe that many, if not most, errors in specifications can be demonstrated using only a small subset of the entire assignment space, a generate-and-test search can be the basis of a practical specification analysis tool.

The second limitation is the time required to generate and test all the assignments. This limitation has far more significant practical implications. Even with a simple specification (such as finder [JD95]) limited to only five underlying objects, the total number of assignments required to generate and test exceeds 10^{27}.

Generating all 10^{27} assignments is clearly inconceivable, rendering naive exhaustive enumeration useless. Fortunately, the vast majority of these assignments are in some sense "duplicates" of other assignments. One assignment may be a permutation of another assignment. Or two assignments may share some common partial assignment, which itself determines the interpretation of the formula. Regardless of the nature of the duplication, generating only one assignment from each set of duplicate assignments is sufficient.

Selective enumeration is a generate-and-test search method that prevents the generation of most duplicates. By preventing the generation of these duplicates, selective enumeration is effective in solving many interesting relational formulae.

1.2 Alloc — An Example

This section introduces a very simple relational specification, which will be used to illustrate points throughout the remainder of this paper. This simple example describes a heap allocation system, such as malloc, in very general terms.

The specification is written in NP [JD96a], a relational specification language that is roughly a subset of Z. NP is limited to first-order objects, so, for example, there are no functions of functions. Figure 1 contains the NP specification for the heap allocation system.

The first line of the example introduces the two *given types* used in this specification, `Addr` and `Value`. A given type is a set of elements, with each element having no internal structure. Every element is contained in exactly one given type. All variables and expressions in NP are typed, indicating that they refer to one of three kinds of values. A variable or expression may refer to (1) an element of a given type, (2) a set of elements of a single given type, or (3) a relation that maps elements of a given type (the domain) to elements of a given type (the range). Relations can be restricted to functions, injections or bijections and they can be restricted to total or surjective relations.

When using NP, specifiers describe their system using a collections of *schemas*, which allow a simple structuring and composition of individual pieces of the specification, similar to the mechanism provided by Z. There are two independent characteristics that jointly classify schemas in NP. A schema is either a *definition*, which defines the system being specified, or a *claim*, which makes assertions about the system being specified. A schema, whether a definition or a claim, refers either to a single state or to a transition between two states. A transitional schema is called an *operation* and describes both a pre-state and a post-state. The specification given in Figure 1 contains examples of three of the four possible combinations of these characteristics, as explained in the following paragraphs.

All schemas have the same basic structure. The *body* of the schema comes after the name of the schema and is enclosed in square brackets (`[ ]`). The body is separated into two sections by a single vertical bar (`|`). The first section defines the variables used in the schema, whereas the second section gives a collection of relational formulae that must all be satisfied in any system described by this specification.

In the example given in Figure 1, Heap is a definitional schema that describes the basic structure of a heap. Heap introduces two variables, usage and used. The variable usage denotes a function mapping addresses (elements of Addr) to their values (elements of Value). The other variable, used, denotes a set that contains all of the addresses currently in use. Heap also defines

```
[Addr, Value]

Heap =
[
  usage : Addr -> Value
  used : set Addr
|
  /* all currently mapped addresses are used */
  used = dom usage
]

Alloc(addr : Addr) =
[
  Heap
|
  /* Allocating a new address does not change the current allocation */
  used <: usage' = usage
  /* But addr is now mapped (to some value unknown) */
  used' = used U {addr}
 ]

uniqueAddrAlloc::
[
  Heap
  a : Addr
|
  /* A newly allocated address should not have been in use */
  Alloc(a) => a not in used
]
```

Figure 1: A trivial NP specification describing a heap allocation system. Addr and Value are the given types. Heap describes the basic structure being manipulated, Alloc describes an allocation operation, and uniqueAddrAlloc is a claim about the specification.

a single formula that describes a relationship that must hold in all valid heaps: the set of addresses in use is exactly the set of addresses currently mapped, that is, the domain of the function usage.

Alloc is an operation that describes the change in a heap when a new piece of memory is allocated. As Alloc refers to Heap in its declaration section, Alloc inherits all of the variables defined by Heap. Within Alloc, the pre-state is referenced using the simple variable names, whereas the post-state is referenced using primed variables, such as usage'. Operations are indicated by the presence of a (possibly empty) parameter list. The parameter list for Alloc defines a single parameter, addr, which is the newly allocated address.

There are two formulas within Alloc. The first (used <: usage' = usage)[1] guarantees that the allocation does not change any existing mappings. The second formula (used' = used U addr)

indicates that the newly allocated address is now considered to be in use (in addition to any addresses already in use).

The third schema, uniqueAddrAlloc, is a claim that asserts that the newly allocated address is not in use prior to the allocation.

1.3 Reducing the Search to Validate uniqueAddrAlloc

A common analysis of NP specifications is to attempt to validate claims such as uniqueAddrAlloc. A claim is *valid* if there are no assignments that satisfy the negation of the claim. Nitpick [JD96b], the tool that I have implemented to analyze relational specifications, validates claims (within user specified finite bounds) using selective enumeration to solve the negation of the claim. The satisfying assignments for the negation of the claim are *counterexamples* of the claim itself.

Selective enumeration recognizes and exploits two basic kinds of duplications: partial assignment duplicates and permutation duplicates. Two assignments are *partial assignment duplicates* if they share a common mapping of values for a subset of the variables (called a *partial assignment*) and that partial assignment itself determines the value of the formula.

The simplest way of exploiting partial assignment duplicates is by exploiting *derived variables* [JD95]. In many specifications, the values of some variables are defined constructively, that is, their value is constrained to be equal to a function of the values of the other variables. Variables with a constructive definition are called derived variables. Given the bindings for the other variables, the search can directly construct the value of a derived variable, rather than generating many possible values and testing each one. For example, the formula used = dom usage appears in the schema Heap. Therefore, the value of used must be exactly the domain of the value of usage in any counterexample to uniqueAddrAlloc. Assuming that usage is bound prior to used being generated, the value of used can be directly computed.

As is obvious from this example, selective enumeration requires the imposition of a variable ordering. Although any ordering is legal for selective enumeration, some orderings yield a much greater reduction in the number of assignments generated than indicated by other orderings.[2]

Because of the constraint a not in used, the value of a must be an element of the value of used for any counterexample to uniqueAddrAlloc. Although this constraint does not limit the possible values of a to a single value, the constraint can be used to limit the values actually generated during the search. *Bounded generation* uses constraints from the formula to limit the values generated. Assuming that used is bound before the value of a is generated, bounded generation will generate each element in the set that is the value of used, instead of each value in the given type Addr.

A second opportunity for bounded generation exists in uniqueAddrAlloc. The first formula in Alloc, used <: usage' = usage, must be true in any counterexample to uniqueAddrAlloc. To

---

1.The <: operator is the domain restriction operator. The result of this expression is a relation that includes all of the pairs in the relation given as the second argument whose first element is contained in the set given as the first argument. For the formal definition of this and other operators, refer to Definition 16 on page 14.

2. The mechanism for choosing an ordering is beyond the scope of this paper. Nitpick uses a heuristic to choose the ordering, as computing an optimal ordering is factorial in the number of variables.

simplify the implementation, bounded generation does not directly take advantage of this constraint; instead, this constraint implies a weaker constraint, usage <= usage. Bounded generation uses this weaker constraint to limit both the domain and range of any value generated for usage to be subsets of the domain and range of the value of usage'.

Derived variable analysis and bounded generation cannot fully exploit all formulae within a specification. If these formulae do not depend on all the variables, they still present an opportunity for reducing the assignments to be generated. *Short circuiting* [DJ96] does not reduce the number of values generated for any variables involved in the formula, as would bounded generation. Instead, short circuiting prevents generation of values for any subsequent variables when the partial assignment cannot satisfy the formula.

An example of short circuiting can be found for the constraint on usage and usage' that initiated the second bounded generation example. Although bounded generation will guarantee that dom usage <= dom usage' and ran usage <= ran usage', this constraint does not guarantee that usage <= usage'. Once usage and usage' have both been generated, short circuiting evaluates the constraint usage <= usage' for the resulting partial assignment. Short circuiting will terminate the current path of the search for any partial assignments not satisfying the constraint. Similarly, once usage, usage', and used have been generated, short circuiting will check the full constraint,
used <: usage' = usage. By utilizing all three techniques, selective enumeration can eliminate all partial assignment duplicates available with the selected ordering.

The second form of duplication is called *permutation duplication*. Because each element in a given type is unstructured, exchanging a pair of elements throughout an assignment does not change the interpretation of the formula for that assignment. *Isomorph elimination* [JJD96;JJD98] prevents the generation of most[3] values that are permutations of other values already generated.

As an example of isomorph elimination, consider the values generated for usage'. If Addr and Value are limited to three elements apiece, it is necessary to generate 64 ($\#domain^{\#range+1}$) values for the partial function usage' without isomorph elimination. With isomorph elimination, on the other hand, only the following seven values need be generated:

$usage' = \varnothing$
$usage' = \{ (a_0, v_0) \}$
$usage' = \{ (a_0, v_0), (a_1, v_1) \}$
$usage' = \{ (a_0, v_0), (a_1, v_0) \}$
$usage' = \{ (a_0, v_0), (a_1, v_1), (a_2, v_2) \}$
$usage' = \{ (a_0, v_0), (a_1, v_0), (a_2, v_1) \}$
$usage' = \{ (a_0, v_0), (a_1, v_0), (a_2, v_0) \}$

The result of this reduced search is illustrated in Figure 2 and Figure 3. Figure 2 demonstrates the search until the first counterexample is found. When the number of elements in Addr and Value are limited to three apiece, derived variables and bounded generation reduce the search to find the first counterexample from 13,851 assignments to just 3. Figure 3 expands the tree for one more value of usage', exhibiting the further advantages of short circuiting and isomorph elimination.

3. The current implementation of isomorph elimination does not consider all possible permutations. In particular, only products of selected single permutations are considered.

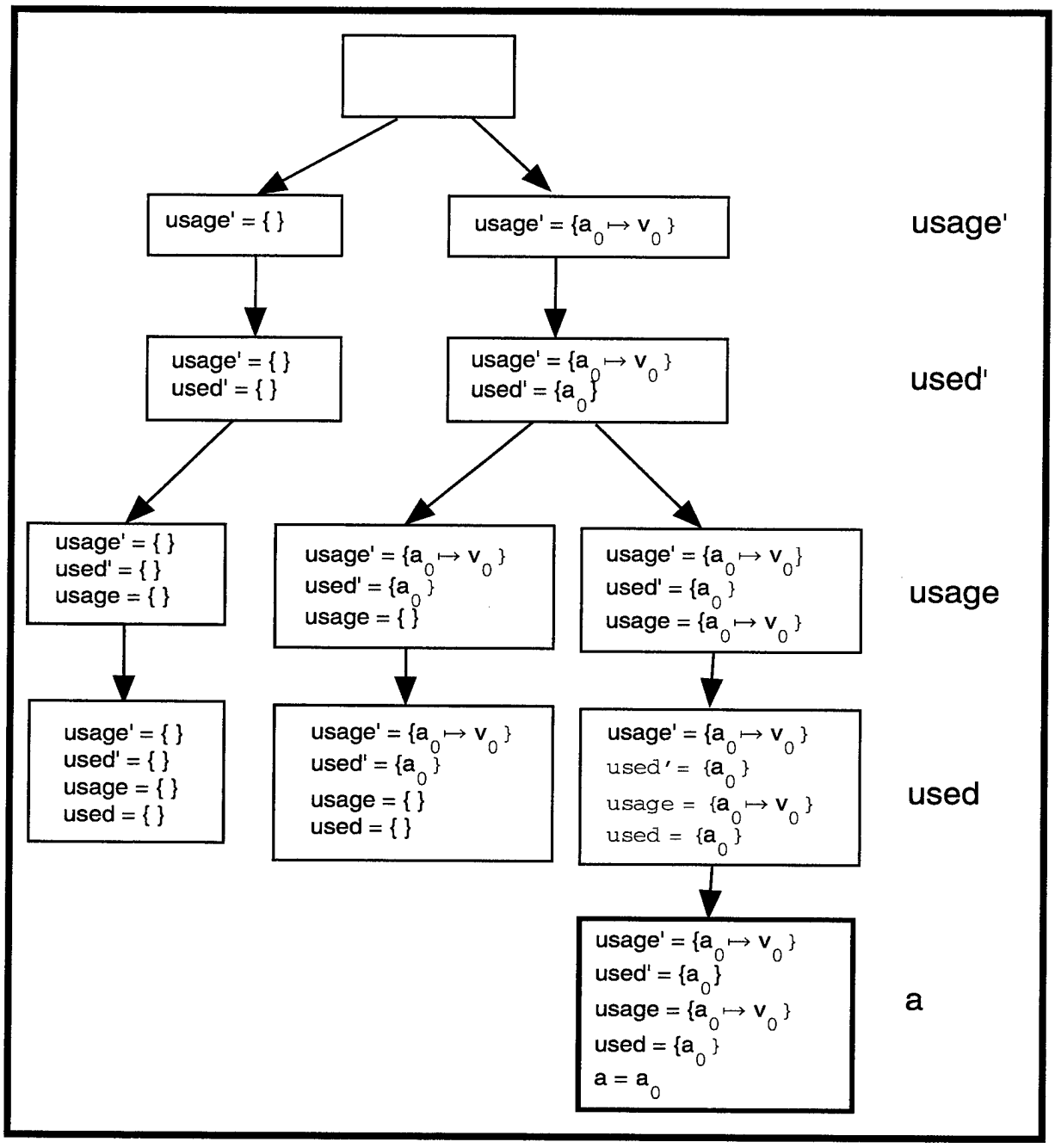

Figure 2: The search tree for finding a counterexample to the claim uniqueAddrAlloc. The variables used' and used are derived; their values can be directly computed from the earlier assignments. Bounded generation limits the domain and range of the values generated for usage to a subset of the domain and range used in usage'. Similarly, bounded generation limits the values considered for a to the elements in the value of used. The first two paths down the search tree result in used being empty, leaving no possible values for a. The first counterexample discovered is shown in a heavier box.

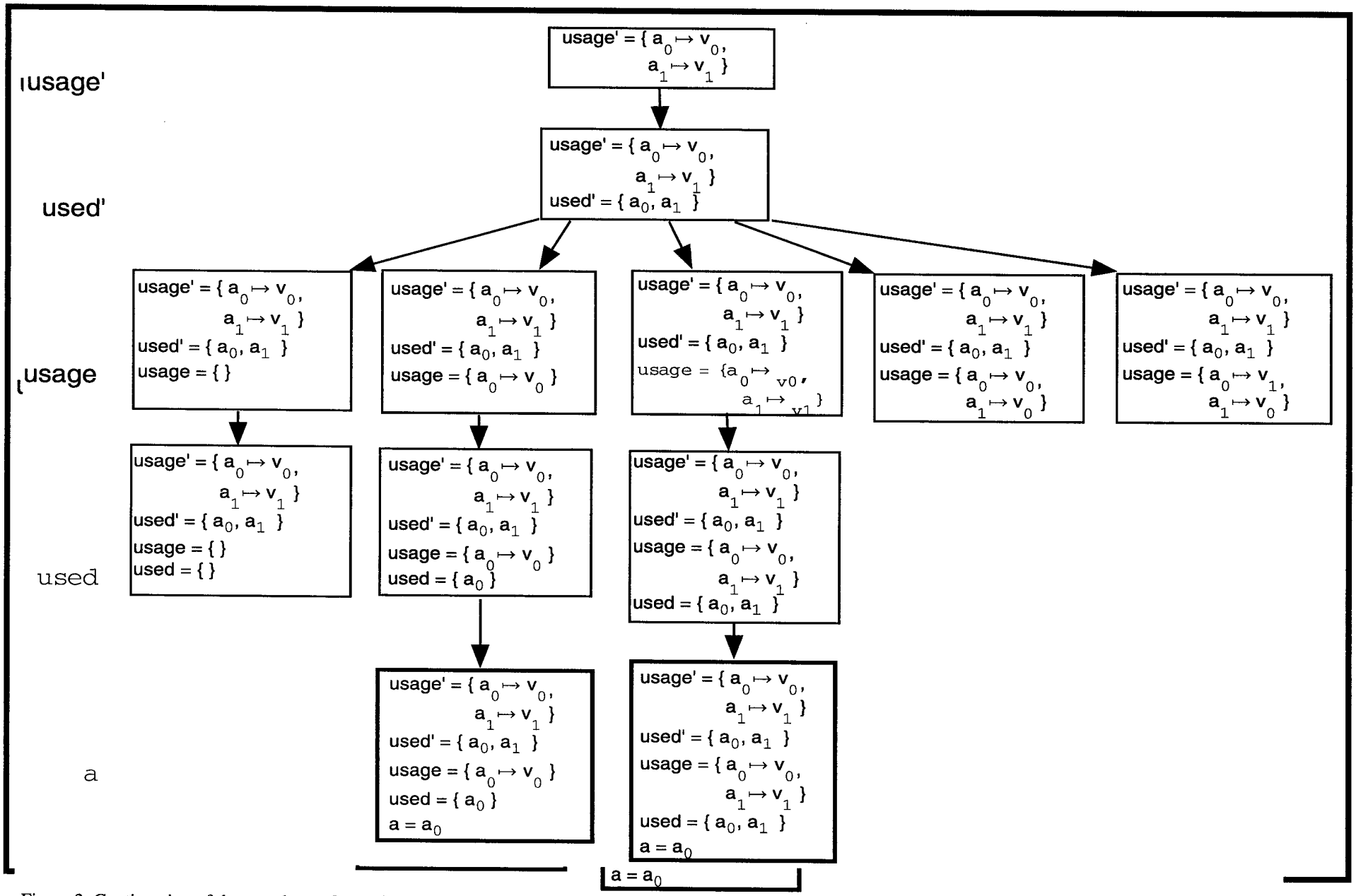

Figure 3: Continuation of the search tree from Figure 2. The satisfying assignments are shown in the heavier boxes. Isomorph elimination generated $\{a_0 \to v_0, a_1 \to v_1\}$ as the next value for usage' because all other single edge values are isomorphic to the one already generated in Figure 2($\{a_0 \to v_0\}$). Short circuiting truncates the search for the rightmost two values generated for usage, as these partial assignments do not satisfy the requirements of alloc. In particular, these partial assignments do not satisfy the formula usage <= usage', which is derived from used <: usage' = usage.

1.4 Related Work

The model-generation community [Zha96; ZZ95; Sla94] has addressed a problem that is similar to the relational formula satisfaction problem addressed by selective enumeration. A model-generation tool searches for a satisfying assignment, or model, for a formula. The logic supported by the model-generation tools varies from the logic supported by NP. Variables are allowed to be arbitrary arity functions in most of these tools, whereas NP directly supports only unary functions. (Arbitrary arity functions can be expressed in NP through careful encoding, but the resultant formula is hard to understand and selective enumeration is particularly inefficient at analyzing such formulae.) NP, on the other hand, adds support for transitive closure, which is difficult or impossible to express generally in the model-generation languages. There is also a difference in apparent goals: selective enumeration is best suited for solving formulae with several variables using a relatively small scope, whereas the model generators appear targeted towards formulae with few variables (frequently one) using a larger scope.

Despite these differences, some of the model generators use an approach similar to the one used in selective enumeration. Zhang [Zha96], in particular, used a similar set of reduction techniques in FALCON. FALCON includes a simpler form of isomorph elimination, a direct equivalent to derived-variable construction and a backtracking feature that is similar to short circuiting. Slaney [Sla94] also uses a backtracking approach in Finder; he achieves reductions similar to those gained with bounded generation by separating the enumeration of functions into separate boolean variables, each representing a single maplet.

Jipsen's approach [Jip92] finds a Boolean algebra with operators (BAO) that satisfies a set of first-order equations. His approach does not require finite bounds, and thus can be used as a true verifier. As the relational calculus can be embedded in BAO, this approach will also solve relational formulae. However, there are a number of difficulties with Jipsen's work. His approach has never been proved complete — it may never terminate for an unsatisfiable system of equations. As transitive closure is not supported within BAO, most of our specifications could not be expressed in full generality. No experimental results are provided, so it is not possible to compare his approach to selective enumeration, even for the finite domain.

The most general related problem is the well known boolean-satisfiability problem. Although the problem itself is NP-complete, researchers have taken two major approaches to achieve an effective solution in realistic time for many formulae. One approach, found in [SLM92] among others, provides an unsound solver, which may fail to return a solution even if one exists. In the other widespread approach, a structure or algorithm provides significantly reduced exponential growth for common formulae, although the worst-case performance may lag even the most naive approach. Binary decision diagrams (or BDDs) [Bry92] are a popular structure that provides this generally reduced exponential growth.

There is a straightforward translation from relational formulae to boolean formulae, so any of the boolean satisfiability approaches can be applied to solve any relational formulae. This conversion loses much of the higher level semantics of the relational formulae. Selective enumeration uses these semantics to produce its reductions. As given in [DJJ96], translating a relational formula into the corresponding boolean one and solving the boolean formula using BDDs required approximately the same time as solving the relational formula directly using selective enumeration. The introduction of bounded generation and an improved isomorph-elimination technique has moved the balance significantly towards selective enumeration. Although it is possible that further effort on the BDD version could similarly reduce the time required, I believe that the additional semantics available in the relational formula will give an advantage to selective enumeration.

Solving a relational formula could be structured as a constraint satisfaction problem. Traditional

constraint satisfaction approaches [Kum92; Mac92], used to solve problems such as shape recognition [Wal75] or job shop scheduling [SF91], support only a much more limited constraint language. Finite constraint satisfaction, the area most similar to selective enumeration, allows only a restricted subset of Horn clauses to express the constraints. For most of the existing constraint satisfaction algorithms, all constraints must be binary (involve no more than two variables). Constraint satisfaction algorithms also typically require a complete enumeration of possible values for each variable, which can be prohibitively expensive for the relation-typed variables commonly found in NP specifications.

There are significant similarities between the approaches taken in constraint satisfaction and the approach taken in selective enumeration. Backtracking in constraint satisfaction is a direct equivalent of short circuiting in selective enumeration, requiring the same care in selecting a variable ordering. The standard constraint propagation algorithms, including arc consistency and k-consistency, are strongly reminiscent of bounded generation. They are limited, however, to the weaker constraint language.

2. Basic Definitions

This section develops the basic terminology for defining selective enumeration precisely. In the following sections, I use this terminology to define each technique that implements selective enumeration. I also prove the soundness of each of these techniques.

This section begins by defining the basic concepts underlying any generate-and-test search. From these, I develop precise definitions of the generators and duplications that are the essence of selective enumeration. A formal definition of soundness follows naturally from these definitions.

2.1 Values and Variables

The search chooses values constructed from the finite universe $\mathcal{U}$ of atomic elements. Each element within $\mathcal{U}$ is itself unstructured. In this initial analysis, I ignore the type distinctions between elements. Therefore, for the simple alloc example described in the prior section, $\mathcal{U}$ is the union of the Addr and Value sets. In Section 6, I will re-introduce given types to differentiate the elements of $\mathcal{U}$.

There are three kinds of values: (1) atomic elements of $\mathcal{U}$, (2) sets of atomic elements, or (3) binary relations on the atomic elements.

Definition 1: $\mathrm{Value}_{\mathrm{scalar}} = \mathcal{U}$

$\mathrm{Value}_{\mathrm{set}} = \mathbb{P}\,\mathcal{U}$

$\mathrm{Value}_{\mathrm{rel}} = \mathbb{P}\,(\mathcal{U} \times \mathcal{U})$

$\mathrm{Value} = \mathrm{Value}_{\mathrm{scalar}} \cup \mathrm{Value}_{\mathrm{set}} \cup \mathrm{Value}_{\mathrm{rel}}$

Each claim or schema in a specification defines a set of variables. I divide the complete collection of variables into three sets based on the kind of value they denote: $\mathrm{Var}_{\mathrm{scalar}}$, $\mathrm{Var}_{\mathrm{set}}$, and $\mathrm{Var}_{\mathrm{rel}}$. The intuitive relationship between these variables and the corresponding values will be maintained; a variable in $\mathrm{Var}_{\mathrm{scalar}}$ will only be bound to an atomic element whereas a variable in $\mathrm{Var}_{\mathrm{set}}$ will only be bound to a set of atomic elements.

Definition 2: $\mathrm{Variable} = \mathrm{Var}_{\mathrm{scalar}} \cup \mathrm{Var}_{\mathrm{set}} \cup \mathrm{Var}_{\mathrm{rel}}$

Definition 3: $N = |\mathrm{Variable}|$

For the claim uniqueAddrAlloc from Figure 1, N is 5 and the variables are

$\mathrm{Var}_{\mathrm{scalar}} = \{\, a \,\}$

$\mathrm{Var}_{\mathrm{set}} = \{\, \mathsf{used}, \mathsf{used}' \,\}$

$\mathrm{Var}_{\mathrm{rel}} = \{\, \mathsf{usage}, \mathsf{usage}' \,\}$

To perform the search, the variables need to be ordered. This order corresponds to the ordering used in the search tree.

Definition 4: $\mathrm{Ord}: \mathrm{Variable} \rightarrow 1 \ldots N$

For the uniqueAddrAlloc example, I will use the ordering from the search illustrated in Figure 2 and Figure 3:

$$Ord = \{ usage' \mapsto 1, used' \mapsto 2, usage \mapsto 3, used \mapsto 4, a \mapsto 5 \}$$

A useful construct is the ordering-based subsets of variables.

Definition 5: $Var_i = \{ v \mid 1 \leqslant Ord(v) \leqslant i \}$

By convention, Var_0 is the empty set of variables. Using the Ord for uniqueAddrAlloc from the prior paragraph, the value for Var_3 is

$$Var_3 = \{ usage', used', usage \}$$

2.2 Language

The next basic element is the language used to express the formula itself. The NP language is intended for human consumption; structures such as schemas offer no additional expressive power. Nitpick translates each specification from the NP language into a simpler formula language. To further simplify this analysis, I define only a subset of the formula language here. Extending this analysis to include the entire formula language is a straightforward exercise.

The alphabet of the simplified formula language includes the variables and the operators.

$$\mathcal{A} = Variable \cup \{ \{, \}, dom, ran, func, U, \&, <:, in, =, <=, Un, (,), and, or, not \}$$

The foundation of the language is the terms. Terms describe all of the values that can be constructed using this language. As with other items, terms are divided into three categories: $Term_{scalar}$, $Term_{set}$, and $Term_{rel}$.

Definition 6: $Term = Term_{scalar} \cup Term_{set} \cup Term_{rel}$, where
$Term_{scalar}$, $Term_{set}$, and $Term_{rel}$ are defined by the BNF grammars

$Term_{scalar}$ ::= Var_{scalar}

$Term_{set}$ ::= $Var_{set} \mid \{ Term_{scalar} \} \mid \{ \} \mid Un \mid$
$(Term_{set} \cup Term_{set}) \mid (Term_{set} \& Term_{set}) \mid$
$(Term_{set} \setminus Term_{set}) \mid Term_{rel} (Term_{scalar}) \mid$
$dom\ Term_{rel} \mid ran\ Term_{rel}$

$Term_{rel}$ ::= $Var_{rel} \mid (Term_{set} <: Term_{rel})$

As an overview, Un is the universe of possible values of the appropriate type, & is the intersection operator, and $Term_{rel}$ ($Term_{scalar}$) gives the relational image. The complete, formal definitions of the operators are given in the definitions starting on page 13.

Atomic formulae are constructed from terms.

Definition 7: AtomicFormula, the set of atomic formulae, is defined by the BNF grammar

AtomicFormula ::= $Term_{scalar}$ in $Term_{set} \mid Term_{scalar} = Term_{scalar} \mid$
$Term_{set} = Term_{set} \mid Term_{set} <= Term_{set} \mid$
$Term_{rel} = Term_{rel} \mid func\ Term_{rel}$

Finally, Wffs in the formula language are built from atomic formulae.

Definition 8: Wff, the set of well-formed formulae, is defined by the BNF grammar

Wff ::= AtomicFormula | not Wff → Wff | (Wff and Wff) | (Wff or Wff)

For any formula in the language, there is a unique derivation for that formula given by this grammar. The claim uniqueAddrAlloc from Figure 1, which is written in NP, is translated into the formula language as

Formula 1:
```
( not ( (dom usage = used and dom usage' = used')
        and ( func usage and func usage' ) ) or
(not ( (dom usage = used and dom usage' = used') and
        ( (used <: usage') = usage and used' = (used U {a}) ) )
or not a in used) )
```

For the formula language, the set of free variables for a formula is exactly the set of variables used in the formula.

Definition 9: The free variables of a term, $FV(\tau) : Term \rightarrow \mathbb{P}Variable$, is defined as

if τ is v where $v \in Variable$
$\{ v \}$

if τ is $\{ \tau_1 \}$ where $\tau_1 \in Term_{scalar}$
$FV(\tau_1)$

if τ is { }
$\varnothing$

if τ is Un
$\varnothing$

if τ is $(\tau_1 \text{ op } \tau_2)$ where $\tau_1, \tau_2 \in Term$ and op is one of U , &, <:, or \
$FV(\tau_1) \cup FV(\tau_2)$

if τ is op τ_1 where $\tau_1 \in Term_{rel}$ and op is either dom or ran
$FV(\tau_1)$

if τ is τ_1 (τ_2) where $\tau_1 \in Term_{rel}$, $\tau_2 \in Term_{scalar}$
$FV(\tau_1) \cup FV(\tau_2)$

Definition 10: The free variables of a formula, $FV(\phi) : Wff \rightarrow \mathbb{P}Variable$, is defined as

if ϕ is τ_1 op τ_2 where $\tau_1, \tau_2 \in Term$ and op is either =, in, or <=
$FV(\tau_1) \cup FV(\tau_2)$

if ϕ is func τ_1 where $\tau_1 \in Term_{rel}$
$FV(\tau_1)$

if ϕ is $(\phi_1 \text{ op } \phi_2)$ where $\phi_1, \phi_2 \in Wff$ and op is either and or or
$FV(\phi_1) \cup FV(\phi_2)$

if ϕ is not ϕ_1 where $\phi_1 \in Wff$
$FV(\phi_1)$

2.3 Assignments

A generate-and-test search solves a formula by generating assignments, then interpreting the formula for each assignment generated. An assignment is a mapping from variables to appropriate values. An assignment can be a full assignment, mapping all variables to appropriate values, or it can be a partial assignment, mapping only a subset of the variables to values.

Definition 11: $S : \text{Variable} \nrightarrow \text{Value} = \{ v \mapsto x \mid$
$v \in \text{Var}_{\text{scalar}} \Rightarrow x \in \text{Value}_{\text{scalar}} \wedge$
$v \in \text{Var}_{\text{set}} \Rightarrow x \in \text{Value}_{\text{set}} \wedge$
$v \in \text{Var}_{\text{rel}} \Rightarrow x \in \text{Value}_{\text{rel}} \}$

S, therefore, is the set of all well-typed assignments. A useful decomposition of S is based on what variables are actually mapped.

Definition 12: $S_i = \{ s \in S \mid \text{dom } s = \text{Var}_i \}$

S_i is the set of all assignments that map exactly the first i variables, as defined by Ord. The assignments on the i^{th} level of the search tree are drawn from S_i. Two such sets are of particular interest: S_0 contains only the empty assignment and S_N contains all full assignments.

2.4 Interpretation

A precise semantics of the formula language is needed to analyze the assignments. A formula is interpreted as true, false or unknown for any given assignment. Intuitively, the interpretation of a formula may be unknown if any of the free variables of the formula are not mapped by the assignment. Otherwise, each variable in the formula is replaced by the corresponding value from the assignment and the formula is evaluated using the usual semantics for relational formulae.

The interpretation of a formula is dependent on the interpretation of each term. The interpretation of a term in the formula language is given as $\bar{s}(\tau)$. $\bar{s}$ is the union of three functions: $\bar{s}_{\text{scalar}}$, $\bar{s}_{\text{set}}$, and $\bar{s}_{\text{rel}}$.

Definition 13: $\bar{s} = \bar{s}_{\text{scalar}} \cup \bar{s}_{\text{set}} \cup \bar{s}_{\text{rel}}$

Definition 14: $\bar{s}_{\text{scalar}}(\tau) : \text{Term}_{\text{scalar}} \rightarrow \text{Value}_{\text{scalar}} \cup \{ \text{unknown} \} =$

if $\tau \in \text{Var}_{\text{scalar}}$	
$s(\tau)$	if $\tau \in \text{dom } s$
unknown	otherwise

Definition 15: $\bar{s}_{\text{set}}(\tau) : \text{Term}_{\text{set}} \rightarrow \text{Value}_{\text{set}} \cup \{ \text{unknown} \} =$

if τ is v where $v \in \text{Var}_{\text{set}}$	
$s(v)$	if $v \in \text{dom } s$
unknown	otherwise
if τ is $\{ \tau_1 \}$ where $\tau_1 \in \text{Term}_{\text{scalar}}$	
$\{ x \mid x = \bar{s}_{\text{scalar}}(\tau_1) \}$	if $\bar{s}_{\text{scalar}}(\tau_1) \neq \text{unknown}$
unknown	otherwise

if τ is { }
  $\varnothing$

if τ is Un
  $\mathcal{U}$

if τ is $\tau_1 \cup \tau_2$ where $\tau_1,\tau_2 \in \mathrm{Term}_{set}$

$\{ x \mid x \in \bar{s}_{set}(\tau_1) \vee x \in \bar{s}_{set}(\tau_2) \}$	if $\bar{s}_{set}(\tau_1) \neq$ unknown $\wedge$ $\bar{s}_{set}(\tau_2) \neq$ unknown
unknown	otherwise

if τ is $\tau_1 \& \tau_2$ where $\tau_1,\tau_2 \in \mathrm{Term}_{set}$

$\{ x \mid x \in \bar{s}_{set}(\tau_1) \wedge x \in \bar{s}_{set}(\tau_2) \}$	if $\bar{s}_{set}(\tau_1) \neq$ unknown $\wedge$ $\bar{s}_{set}(\tau_2) \neq$ unknown
unknown	otherwise

if τ is $\tau_1 \setminus \tau_2$ where $\tau_1,\tau_2 \in \mathrm{Term}_{set}$

$\{ x \mid x \in \bar{s}_{set}(\tau_1) \wedge x \notin \bar{s}_{set}(\tau_2) \}$	if $\bar{s}_{set}(\tau_1) \neq$ unknown $\wedge$ $\bar{s}_{set}(\tau_2) \neq$ unknown
unknown	otherwise

if τ is dom τ_1 where $\tau_1 \in \mathrm{Term}_{rel}$

$\{ x \mid \exists y.(x,y) \in \bar{s}_{rel}(\tau_1) \}$	if $\bar{s}_{rel}(\tau_1) \neq$ unknown
unknown	otherwise

if τ is ran τ_1 where $\tau_1 \in \mathrm{Term}_{rel}$

$\{ y \mid \exists x.(x,y) \in \bar{s}_{rel}(\tau_1) \}$	if $\bar{s}_{rel}(\tau_1) \neq$ unknown
unknown	otherwise

if τ is $\tau_1(\tau_2)$ where $\tau_1 \in \mathrm{Term}_{rel} \wedge \tau_2 \in \mathrm{Term}_{scalar}$

$\{ y \mid (\bar{s}_{scalar}(\tau_2),y) \in \bar{s}_{rel}(\tau_1) \}$	if $\bar{s}_{rel}(\tau_1) \neq$ unknown $\wedge$ $\bar{s}_{scalar}(\tau_2) \neq$ unknown
unknown	otherwise

Definition 16: $\bar{s}_{rel}(\tau) : \mathrm{Term}_{rel} \rightarrow \mathrm{Value}_{rel} \cup \{$ unknown $\} =$

if $\tau \in \mathrm{Var}_{rel}$

$s(\tau)$	if $\tau \in$ dom s
unknown	otherwise

if τ is $\tau_1 <: \tau_2$ where $\tau_1 \in \mathrm{Term}_{set} \wedge \tau_2 \in \mathrm{Term}_{rel}$

$\{ (x,y) \mid x \in \bar{s}_{set}(\tau_1) \wedge (x,y) \in \bar{s}_{rel}(\tau_2) \}$	if $\bar{s}_{set}(\tau_1) \neq$ unknown $\wedge$ $\bar{s}_{rel}(\tau_2) \neq$ unknown
unknown	otherwise

The three $\bar{s}$ functions derive from directly s. Therefore, each assignment s induces a corresponding $\bar{s}$ function.

The interpretation of a formula for an assignment is given using $\models\phi[s]$, where $s \in S$ and $\phi \in$ Wff. It follows from the $\bar{s}$ functions.

Definition 17: $\models\phi[s] : S \times \mathrm{Wff} \rightarrow \{$ true, false, unknown $\} =$

if ϕ is $\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in \mathrm{Term}_{\mathrm{scalar}}$

true	if $\bar{s}_{\mathrm{scalar}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{scalar}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{scalar}}(\tau_1) = \bar{s}_{\mathrm{scalar}}(\tau_2)$
false	if $\bar{s}_{\mathrm{scalar}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{scalar}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{scalar}}(\tau_1) \neq \bar{s}_{\mathrm{scalar}}(\tau_2)$
unknown	otherwise

if ϕ is τ_1 in τ_2 where $\tau_1 \in \mathrm{Term}_{\mathrm{scalar}}$ and $\tau_2 \in \mathrm{Term}_{\mathrm{set}}$

true	if $\bar{s}_{\mathrm{scalar}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{scalar}}(\tau_1) \in \bar{s}_{\mathrm{set}}(\tau_2)$
false	if $\bar{s}_{\mathrm{scalar}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{scalar}}(\tau_1) \notin \bar{s}_{\mathrm{set}}(\tau_2)$
unknown	otherwise

if ϕ is $\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in \mathrm{Term}_{\mathrm{set}}$

true	if $\bar{s}_{\mathrm{set}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_1) = \bar{s}_{\mathrm{set}}(\tau_2)$
false	if $\bar{s}_{\mathrm{set}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_1) \neq \bar{s}_{\mathrm{set}}(\tau_2)$
unknown	otherwise

if ϕ is $\tau_1 <= \tau_2$ where $\tau_1, \tau_2 \in \mathrm{Term}_{\mathrm{set}}$

true	if $\bar{s}_{\mathrm{set}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_1) \subseteq \bar{s}_{\mathrm{set}}(\tau_2)$
false	if $\bar{s}_{\mathrm{set}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{set}}(\tau_1) \not\subset \bar{s}_{\mathrm{set}}(\tau_2) \wedge \bar{s}_{\mathrm{set}}(\tau_1) \neq \bar{s}_{\mathrm{set}}(\tau_2)$
unknown	otherwise

if ϕ is $\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in \mathrm{Term}_{\mathrm{rel}}$

true	if $\bar{s}_{\mathrm{rel}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{rel}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{rel}}(\tau_1) = \bar{s}_{\mathrm{rel}}(\tau_2)$
false	if $\bar{s}_{\mathrm{set}}(\tau_1) \neq \text{unknown} \wedge \bar{s}_{\mathrm{rel}}(\tau_2) \neq \text{unknown} \wedge \bar{s}_{\mathrm{rel}}(\tau_1) \neq \bar{s}_{\mathrm{rel}}(\tau_2)$
unknown	otherwise

if ϕ is func τ_1 where $\tau_1 \in \mathrm{Term}_{rel}$

true	if $\bar{s}_{rel}(\tau_1) \neq$ unknown $\wedge$ $\forall x,y,z \in \mathcal{U}.$ $((x,y) \in \bar{s}_{rel}(\tau_1) \wedge (x,z) \in \bar{s}_{rel}(\tau_1)) \Rightarrow y = z$
false	if $\bar{s}_{rel}(\tau_1) \neq$ unknown $\wedge$ $\forall x,y \in \mathcal{U}.\ \exists z \in \mathcal{U}.y \neq z \wedge$ $((x,y) \in \bar{s}_{rel}(\tau_1) \wedge (x,z) \in \bar{s}_{rel}(\tau_1))$
unknown	otherwise

if ϕ is (ϕ_1 and ϕ_2) where $\phi_1,\phi_2 \in$ Wff

true	if $\models\phi_1[s] =$ true $\wedge \models\phi_2[s] =$ true
false	if $\models\phi_1[s] =$ false $\vee \models\phi_2[s] =$ false
unknown	otherwise

if ϕ is (ϕ_1 or ϕ_2) where $\phi_1,\phi_2 \in$ Wff

true	if $\models\phi_1[s] =$ true $\vee \models\phi_2[s] =$ true
false	if $\models\phi_1[s] =$ false $\wedge \models\phi_2[s] =$ false
unknown	otherwise

if ϕ is not ϕ_1 where $\phi_1 \in$ Wff

true	if $\models\phi_1[s] =$ true
false	if $\models\phi_1[s] =$ false
unknown	otherwise

As an example, consider the negation of Formula 1 (derived from uniqueAddrAlloc) given by Formula 2:

Formula 2: ϕ = (((dom usage = used and dom usage' = used') and
(func usage and func usage')) and
(((used <: usage') = usage and used' = (used U {a}))
and a in used))

This formula can be interpreted using any valid assignment. Assuming a_0 and v_0 are elements of $\mathcal{U}$, three possible interpretations are

$\models\phi[\{\ \}]$ = unknown

$\models\phi[\{$ usage'$\mapsto\{ (a_0, v_0) \}$, used'$\mapsto\varnothing\ \}]$ = false

$\models\phi[\{$ usage'$\mapsto\{ (a_0, v_0) \}$, used'$\mapsto\{a_0\}$,
usage$\mapsto\{ (a_0, v_0) \}$, used$\mapsto\{a_0\}$, a$\mapsto a_0\ \}]$ = true

If the assignment maps all of the free variables in the formula, the interpretation will be either true or false, but not unknown.

Theorem 1: $\forall s \in S, \tau \in \mathrm{Term}.\mathrm{FV}(\tau) \subseteq$ dom s $\Rightarrow \bar{s}(\tau) \neq$ unknown

Proof: By structural induction.

If τ is v where $v \in$ Variable

By definition, $FV(\tau) = \{ v \}$

As $FV(\tau) \subseteq$ dom s, $v \in$ dom s

By definition of $\bar{s}$, $v \in$ dom s $\Rightarrow \bar{s}(\tau) \neq$ unknown.

if τ is $\tau_1 \cup \tau_2$ where $\tau_1, \tau_2 \in Term_{set}$

By definition of FV, $FV(\tau_1) \subseteq FV(\tau)$ and $FV(\tau_2) \subseteq FV(\tau)$

Therefore, $FV(\tau_1) \subseteq$ dom s and $FV(\tau_2) \subseteq$ dom s

Therefore, by induction, $\bar{s}(\tau_1) \neq$ unknown and $\bar{s}(\tau_2) \neq$ unknown

By definition of $\bar{s}$, $\bar{s}(\tau) \neq$ unknown

Other productions follow similarly ■

Theorem 2: $\forall s \in S, \phi \in wff. FV(\phi) \subseteq$ dom s $\Rightarrow \models\phi[s] \neq$ unknown

Proof: By structural induction.

If ϕ is τ_1 in τ_2 where $\tau_1 \in Term_{scalar}$ and $\tau_2 \in Term_{set}$

By definition of FV, $FV(\tau_1) \subseteq FV(\phi)$ and $FV(\tau_2) \subseteq FV(\phi)$

Therefore, $FV(\tau_1) \subseteq$ dom s and $FV(\tau_2) \subseteq$ dom s

Therefore, by Theorem 1, $\bar{s}(\tau_1) \neq$ unknown and $\bar{s}(\tau_2) \neq$ unknown

By definition of $\models$, $\models\phi[s] \neq$ unknown

If ϕ is (ϕ_1 and ϕ_2) where $\phi_1, \phi_2 \in$ Wff

By definition of FV, $FV(\phi_1) \subseteq FV(\phi)$ and $FV(\phi_2) \subseteq FV(\phi)$

Therefore, $FV(\phi_1) \subseteq$ dom s and $FV(\phi_2) \subseteq$ dom s

Therefore, by induction, $\models\phi_1[s] \neq$ unknown and $\models\phi_2[s] \neq$ unknown

By definition of $\models$, $\models\phi[s] \neq$ unknown

Other productions follow similarly ■

Some formulae are logically implied by other formulae. The notation $\phi \models \phi'$ indicates that ϕ logically implies ϕ'.

Definition 18: $\phi \models \phi'$ iff $\forall s \in S_N. \models\phi[s] =$ true $\Rightarrow \models\phi'[s] =$ true

Given the formula ϕ used in the preceding example, $\phi \models$ a in used.

2.5 Generators

The key to any generate-and-test search is the ability to generate assignments. A special function, called a generator, generates assignments for level i of the search tree given an assignment from level i-1. A generator for level i adds a value for the i^{th} variable to the initial assignment, without changing the mapping of any other variable in that assignment.

Definition 19: [4]A function $g_i : S_{i-1} \rightarrow \mathbb{P}S_i$ is a generator for level i of the search iff

$\forall s \in S_{i-1}. \forall s' \in g_i(s). Var_{i-1} \lhd s' = s$

A generator function expands a single assignment in the search tree into the complete set of assignments descending immediately from that value.

An aggregate generator can also be defined for any generator. An aggregate generator generates a complete level in the search tree, given the prior level of the search tree.

Definition 20: A function $G_i : \mathbb{P}S_{i-1} \rightarrow \mathbb{P}S_i$ is an aggregate generator for a level i generator g_i iff

$$\forall Q_{i-1} \subseteq S_{i-1}.\ G_i(Q_{i-1}) = \bigcup_{q \in Q_{i-1}} g_i(q)$$

A trivial generator, corresponding to an exhaustive-enumeration search, can be associated with any variable using the exhaustive-enumeration generator g0.

Definition 21: The exhaustive-enumeration generator g0 for level i is defined as

if $v_i \in Var_{scalar}$

$$g0(s) = \{\ s' \mid \exists x \in Value_{scalar}.\ s' = s \cup \{\ v_i \mapsto x\ \}\ \}$$

if $v_i \in Var_{set}$

$$g0(s) = \{\ s' \mid \exists x \in Value_{set}.\ s' = s \cup \{\ v_i \mapsto x\ \}\ \}$$

if $v_i \in Var_{rel}$

$$g0(s) = \{\ s' \mid \exists x \in Value_{rel}.\ s' = s \cup \{\ v_i \mapsto x\ \}\ \}$$

A search requires a collection of generators, one associated with each variable. A function mapping each variable to an appropriate generator is called a generator suite.

Definition 22: A function γ : Variable $\rightarrow$ (S $\rightarrow \mathbb{P}$S) is a generator suite iff

$\forall v \in$ Variable. Ord(v) = i $\Rightarrow \gamma(v)$ is a generator for level i.

2.6 Duplications

The essence of selective enumeration is reducing the number of cases generated by removing duplicates. A duplication partitions the set of full assignments into equivalence classes for some particular formula ϕ. The only requirement for these equivalence classes is that all assignments in any equivalence class give the same interpretation to ϕ.

Definition 23: A set of sets $d(\phi)$: $\mathbb{P}\mathbb{P}S_N$ is a duplication for the formula ϕ iff

$d(\phi)$ is a partitioning of S_N and

for the equivalence relation $\approx_{d(\phi)}$ induced by $d(\phi)$, $\forall s,s' \in S_N.s \approx_{d(\phi)} s' \Rightarrow \models\phi[s] = \models\phi[s']$

Two obvious duplications are uninteresting for the purposes of selective enumeration. The first, which I call d_0, places each assignment in its own equivalence class. This corresponds to the exhaustive-enumeration search. The second obvious duplication, which I call $d_\models$, divides the

4. I use the domain restriction operator $\lhd$ from Z here as well as in later definitions and theorems. For those readers unfamiliar with Z, the $\lhd$ operator yields the relation that is the subset of the second operand restricted to those pairs whose first element is a member of the first operand. More precisely, $s \lhd r = \{\ (x,y) \mid (x,y) \in r \wedge x \in s\}$. In practice, this is used to select a subset of a relation that is meaningful for a more restricted domain.

assignments into two classes, ones that satisfy ϕ and ones that do not satisfy ϕ. Although this would be the ideal duplication, it is not directly computable and therefore of no great benefit.

These two duplications can be defined in terms of the corresponding equivalence relations.

$$\forall s,s' \in S_N.\ s \approx_0 s' \Leftrightarrow s = s'$$

$$\forall s,s' \in S_N.\ s \approx_{\models\phi} s' \Leftrightarrow \models\phi[s] = \models\phi[s']$$

The duplications described in the previous section for solving uniqueAddrAlloc can also be defined in this manner. For the reduction involving a in used, every assignment for which a was not an element of used was placed into a single equivalence class, with each other assignment defining its own equivalence class.

$$\forall s,s' \in S_N.\ s \approx_{a\ \text{in}\ used} s' \Leftrightarrow$$
$$((\models a\ \text{in}\ used[s] = false \wedge \models a\ \text{in}\ used[s'] = false) \vee s = s')$$

Other bounded generation duplications, such as the constraint on usage and usage', behave similarly. Each duplication groups known false assignments together in a single equivalence class, placing all other assignment into individual equivalence classes.

This form of equivalence relation can be generalized to support any partial assignment duplication. Each partial assignment duplication has a related formula ϕ' that is implied by the target formula itself. It is convenient to define a special notation to describe partial assignment duplications.

Definition 24: An equivalence relation $\approx_{PAd(\phi,\phi')}$ is a partial assignment duplicate equivalence relation iff $\phi \models \phi' \wedge \forall s,s' \in S_N.s \approx_{PAd(\phi,\phi')} s' \Leftrightarrow (s = s' \vee (\models\phi'[s] = false \wedge \models\phi'[s'] = false))$

This places all assignments that fail to satisfy ϕ' into a single equivalence class and each assignment that satisfies ϕ' into its own equivalence class.

Theorem 3: The partitioning of S_N induced by a partial assignment duplicate equivalence relation $\approx_{PAd(\phi,\phi')}$ is a duplication for ϕ.

Proof: To prove that $\approx_{PAd(\phi,\phi')}$ induces a duplication, it is necessary to prove that

$$\forall \phi{:}Wff.\forall s,s' \in S_N.s \approx_{PAd(\phi,\phi')} s' \Rightarrow \models\phi[s] = \models\phi[s']$$

By the definition of $\approx_{PAd(\phi,\phi')}$, there are two cases to consider:

$$s = s' \text{ and } (\models\phi'[s] = false \wedge \models\phi'[s'] = false)$$

For the first case, clearly $\models\phi[s] = \models\phi[s']$.

For the second case, if $s \in S_N$, $\models\phi[s]$ is either true or false by Theorem 2.
Because $\phi \models \phi'$, $\models\phi[s] = true \Rightarrow \models\phi'[s] = true$.
Therefore, if $\models\phi'[s] = false$, $\models\phi[s] = false$.
Therefore, for the second case, $\models\phi[s] = false \wedge \models\phi[s'] = false$. ■

Definition 25: $PAd(\phi,\phi')$ is the partial assignment duplication induced by the partial assignment equivalence relation $\approx_{PAd(\phi,\phi'))}$.

I will not describe the permutation duplications until I formalize the notion of permutations in Section 4.

Duplications may also be combined, further reducing the number of assignments that selective enumeration must generate and test. To combine two duplications, the corresponding equivalence relations are combined.

Definition 26: $\approx_a = \approx_b \circ \approx_c$ iff

$$\forall s,s' \in S_N.s \approx_a s' \Leftrightarrow (s \approx_b s' \vee s \approx_c s' \vee (\exists s'' \in S_N.(s \approx_a s'' \wedge s'' \approx_a s')))$$

The result of combining two equivalence relations is itself an equivalence relation that induces a duplication.

Lemma 4: For any two equivalence relations $\approx_b$ and $\approx_c$, $\approx_b \circ \approx_c$ is an equivalence relation.

Proof: A relation must be reflexive, symmetric and transitive to be an equivalence relation.

As $\approx_b$ is reflexive, $\forall s \in S_N.s \approx_b s$.
Therefore, $\forall s \in S_N.s \approx_{b\circ c} s$.

If $s \approx_{b\circ c} s'$, then either $s \approx_b s'$ or $s \approx_c s'$.
Assuming $s \approx_b s'$, then by symmetry of $\approx_b$, $s' \approx_b s$.
Therefore $s' \approx_{b\circ c} s$ and $\approx_{b\circ c}$ is symmetric.

By the definition of $\approx_b \circ \approx_c$,

$$\forall s,s',s'' \in S_N.s \approx_{b\circ c} s' \wedge s' \approx_{b\circ c} s'' \Rightarrow s \approx_{b\circ c} s''.$$ ■

Definition 27: $d_b(\phi) \circ d_c(\phi)$ is the partitioning induced by $\approx_b \circ \approx_c$

Theorem 5: $d_b(\phi) \circ d_c(\phi)$ is a duplication.

Proof: Let $d_a(\phi) = d_b(\phi) \circ d_c(\phi)$ and $\approx_a = \approx_b \circ \approx_c$.

There are two requirements for $d_a(\phi)$ to be a duplication:
it must be a partitioning of S_N and
$\forall \phi \in Wff, \forall s,s' \in S_N.s \approx_a s' \Rightarrow \models\phi[s] = \models\phi[s']$.

By Lemma 4, $\approx_a$ is an equivalence relation on S_N,
therefore $d_a(\phi)$ is a partitioning of S_N.

Assume $s,s' \in S_N$ such that $s \approx_a s'$
By definition, one of

$$s \approx_b s'$$
$$s \approx_c s'$$
$$\exists s'' \in S_N.(s \approx_a s'' \wedge s'' \approx_a s')$$

must hold.

If either of the first two possibilities hold
then $\models\phi[s] = \models\phi[s']$ by definition of $d_b(\phi)$ or $d_c(\phi)$, respectively.

The third possibility requires the existence of a sequence of full assignments, $s^1, s^2, ..., s^k \in S_N . s \approx^1 s^1 \wedge s^1 \approx^2 s^2 \wedge s^2 \approx^3 s^3 \wedge ... \wedge s^{k-1} \approx^k s^k \wedge s^k \approx^{k+1} s'$ where $\approx^1, \approx^2, ..., \approx^k, \approx^{k+1}$ are either $\approx_b$ or $\approx_c$.

It is obvious by induction that such a sequence must guarantee that $\models\phi[s] = \models\phi[s']$. ■

Combining two partial assignment duplications in this manner gives the same result as the partial assignment duplication defined by the formula built from the conjunction of the two formulae defining the original duplications.

Theorem 6: $PAd(\phi,\phi_1') \circ PAd(\phi,\phi_2') = PAd(\phi,(\phi_1' \text{ and } \phi_2'))$.

Proof: There are two distinct requirements that must be demonstrated equivalent for this theorem to hold.

The first is straightforward:

$$\phi \models \phi_1' \wedge \phi \models \phi_2' \Leftrightarrow \phi \models (\phi_1' \text{ and } \phi_2')$$

Let $\approx_1$ be the equivalence relation induced by $PAd(\phi,\phi_1')$,
$\approx_2$ be the equivalence relation induced by $PAd(\phi,\phi_2')$ and
$\approx_{and}$ be the equivalence relation induced by $PAd(\phi,(\phi_1' \text{ and } \phi_2'))$

The second requirement is that

$$(s \approx_1 s' \vee s \approx_2 s') \Rightarrow s \approx_{and} s' \wedge$$
$$(\exists s'' \in S_N . (s \approx_{and} s'' \wedge s'' \approx_{and} s')) \Rightarrow s \approx_{and} s'.$$

The second clause of this requirement is a direct consequence of the fact that $\approx_{and}$ is an equivalence relation.

For the first clause, if $s = s'$, then $s \approx_1 s'$, $s \approx_2 s'$, and $s \approx_{and} s'$.

Otherwise, if $s \approx_1 s'$, then $\models\phi_1'[s] = \text{false} \wedge \models\phi_1'[s'] = \text{false}$.

Therefore, then $\models(\phi_1' \text{ and } \phi_2')[s] = \text{false} \wedge \models(\phi_1' \text{ and } \phi_2')[s'] = \text{false}$

Therefore, $s \approx_{and} s'$.

Similarly, if $s \approx_2 s'$, then $s \approx_{and} s'$. ■

2.7 Soundness

It is now possible to define a selective enumeration search precisely.

Definition 28: A function $\omega : Wff \times GeneratorSuite \rightarrow \mathbb{P}S_N$ is a search iff

$$\omega(\phi,\gamma) = \{ s \mid \models\phi[s] = \text{true} \wedge s \in G_N(G_{N-1}(G_{N-2}(...G_2(G_1(\{\varnothing\}))))) \}$$

where G_i is the aggregate generator for each g_i in γ.

To be sound, a search must guarantee that it will find at least one satisfying assignment if any exist.

Definition 29: A search $\omega(\phi,\gamma)$ is sound iff $(\exists s \in S_N . \models\phi[s] = \text{true}) \Rightarrow \omega(\phi,\gamma) \neq \varnothing$.

For any duplication $d(\phi)$ for a formula ϕ, selective enumeration will be sound if it enumerates at least one assignment from each satisfying set in $d(\phi)$. A generator is sound if the only satisfying assignments excluded are duplicates of other assignments generated.

Definition 30: A level i generator g_i is sound for a duplication $d(\phi)$ iff

$$\forall s \in S_N.\models\phi[s] = \text{true} \Rightarrow \exists s' \in S_N.Var_i \lhd s' \in g_i(Var_{i-1} \lhd s) \wedge s \approx_{d(\phi)} s'.$$

A subset of S_N is said to represent $d(\phi)$ if it includes at least one assignment from each satisfying set in $d(\phi)$. This can be easily extended to include subsets of S_i that can be used to generate a representative set .

Definition 31: A set $Q_i \subseteq S_i$ represents $d(\phi)$ iff

$$\forall s \in S_N.(\models\phi[s] = \text{true} \Rightarrow \exists s' \in S_N.(s \approx_{d(\phi)} s' \wedge Var_i \lhd s' \in Q_i)).$$

As a base case, the set containing only the empty assignment represents any duplication.

Lemma 7: $\{ \varnothing \} \subseteq S_0$ represents any $d(\phi)$.

Proof: Proof by construction.

By definition, $Var_0 = \varnothing$.

Therefore, $\forall s \in S_N.Var_0 \lhd s = \varnothing$. ■

Starting with a representative set, a sound generator will generate a representative set.

Theorem 8: If a level i generator g_i with the aggregate generator G_i is sound for $d(\phi)$ then

$$\forall Q_{i-1} \subseteq S_{i-1}.Q_{i-1} \text{ represents } d(\phi) \Rightarrow G_i(Q_{i-1}) \text{ represents } d(\phi).$$

Proof: By contradiction.

Assume $s \in S_N.\models\phi[s] = \text{true} \wedge \forall s' \in S_N.(s \approx_{d(\phi)} s' \Rightarrow Var_i \lhd s' \notin G_i(Q_{i-1}))$.

By definition of represents,

$$\exists s'' \in S_N.(s \approx_{d(\phi)} s'' \wedge Var_{i-1} \lhd s'' \in Q_{i-1})).$$

Because $\models\phi[s] = \text{true}$ and g_i is sound for $d(\phi)$,

$$\exists s''' \in S_N.(s'' \approx_{d(\phi)} s''' \wedge Var_i \lhd s''' \in G_i(Q_{i-1})).$$

By transitivity, $s \approx_{d(\phi)} s'''$.

Therefore, the assumption is contradicted. ■

A combination of duplications defines a new collection of equivalence classes that completely includes all of the original equivalence classes. Therefore, a generator that is sound for one duplication is also sound for that duplication in combination with any other duplication.

Theorem 9: If a level i generator g_i is sound for any duplication $d_1(\phi)$,
for any duplication $d_2(\phi)$, g_i is sound for $d_1(\phi) \circ d_2(\phi)$.

Proof: Let $s \in S_N.\models\phi[s] = \text{true}$,

$\approx_1$ be the equivalence relation induced by $d_1(\phi)$,

$\approx_{1\circ 2}$ be the equivalence relation induced by $d_1(\phi) \circ d_2(\phi)$.

By the definition of a sound generator,

$$\exists s' \in S_N.\mathrm{Var}_i \lhd s' \in g_i(\mathrm{Var}_{i-1} \lhd s) \wedge s \approx_1 s'.$$

By definition of $\circ$, $s \approx_1 s' \Rightarrow s \approx_{1\circ 2} s'$.

Therefore, g_i is sound for $d_1(\phi) \circ d_2(\phi)$. ■

For a sound search, the goal is a collection of generators that can generate a representative set of full assignments.

Lemma 10: The result of a search $\omega(\phi,\gamma)$ represents $d(\phi)$ if $\forall\, g_i \in \mathrm{ran}\ \gamma.\ g_i$ is sound for $d(\phi)$.

Proof: By induction on i.

Hypothesis: $G_i(G_{i-1}(...(G_1(\{\varnothing\}))))$ represents $d(\phi)$ if $\forall g_i \in \mathrm{ran}\ \gamma.g_i$ is sound for $d(\phi)$.

By Lemma 7,$\{\varnothing\}$ represents $d(\phi)$.
Therefore, because g_1 is sound for $d(\phi)$, $G_1(\{\varnothing\})$ represents $d(\phi)$ by Theorem 8.

By induction, $G_{i-1}(G_{i-2}(...(G_1(\{\varnothing\}))))$ represents $d(\phi)$.
Because g_i is sound for $d(\phi)$,

$G_i(G_{i-1}(...(G_1(\{\varnothing\}))))$ represents $d(\phi)$ by Theorem 8. ■

Theorem 11: A search $\omega(\phi,\gamma)$ is sound if $\exists\, d(\phi).\ \forall\, g_i \in \mathrm{ran}\ \gamma.\ g_i$ is sound for $d(\phi)$.

Proof: Assume an assignment s exists such that $\models\phi[s] = \mathrm{true}$.

By Lemma 10, $\omega(\phi, \gamma)$must represent $d(\phi)$.
By definition of represents,

$$\exists q \in \omega(\phi, \gamma).\exists\, s' \in S_N .q = \mathrm{Var}_i \lhd s' \wedge s \approx_{d(\phi)} s'.$$

Because $\omega(\phi, \gamma) \subseteq S_N$ and $\mathrm{Var}_N \lhd s' = s'$, $q = s'$.
This implies that $s' \in \omega(\phi, \gamma)$. ■

3. Bounded Generation

In the first section, I introduced bounded generation by means of a few examples. In this section, I clarify what is meant by bounded generation and prove that any bounded-generation generator is sound.

3.1 Overview of Bounded Generation

As introduced in the first section, bounded generation limits the values to be generated for any variable by limiting the underlying universe of elements. Using the claim uniqueAddrAlloc from Figure 1, I have demonstrated two concrete examples: constraining the value of a to be an element of the value of used and constraining the domain and range of the value of usage to be subsets of the domain and range of the value of usage'.

The first example demonstrates a direct constraint on the universe of elements from which a scalar value is chosen. In the second example, a relation is restricted to a (generally) smaller universe of elements for the domain and the range. In each example, bounded generation uses information derived from the formula to reduce the possible values to be generated.

This reduction is the essence of bounded generation. Unlike most other techniques, bounded generation does not provide any self-contained generators. Instead, bounded generation depends on other generators, possibly even using generators implementing other reduction techniques.

Another example of bounded generation begins with the atomic formula used' = (used U {a}) found in Formula 2. Any assignment satisfying this formula also satisfies the formula used <= used'. This new formula logically implies used & (Un\used') = {}.

For bounded generation of set variables, such as used, the goal is to divide the universe of elements $\mathcal{U}$ into three distinct subsets: elements that are required to be included in the set denoted by the variable, elements that are never in that set, and elements that may possibly (but not necessarily) be in that set. For this example, any element contained in the set given by $\overline{s}_{set}$(Un\used') can never be in the set denoted by used for any satisfying assignment. With a different value for Ord (generating values for a before generating values for used), it could also be determined that the element denoted by a is in the set denoted by used for any satisfying assignment. This relationship is illustrated in Figure 4.

The set of elements that may possibly be included is described by a term, which I call $\mathcal{P}$. For this example, $\mathcal{P}$ is (used' \ a). The set of elements that are required for any satisfying assignment are described by a similar term, which I call $\mathcal{R}$. For this example, $\mathcal{R}$ is a. The underlying generator needs to consider only the elements contained in $\overline{s}_{set}(\mathcal{P})$. Bounded generation unions each value yielded by the underlying generator with the value of $\overline{s}_{set}(\mathcal{R})$. If the underlying generator generates each subset of $\overline{s}_{set}(\mathcal{P})$, bounded generation will be sound. In fact, as will be shown in the following sections, bounded generation is sound when combined with some underlying generators that do not yield all possible subsets of $\overline{s}_{set}(\mathcal{P})$.

For scalar variables, there is no subset of elements always included, so the value of $\mathcal{R}$ is always {}. A naive underlying generator could yield each element of $\overline{s}_{set}(\mathcal{P})$, which the bounded-generation generator would yield unchanged.

Relational variables are the most complicated for bounded generation. The most consistent and efficient definition of bounded generation for relations would consider sets of edges, rather than

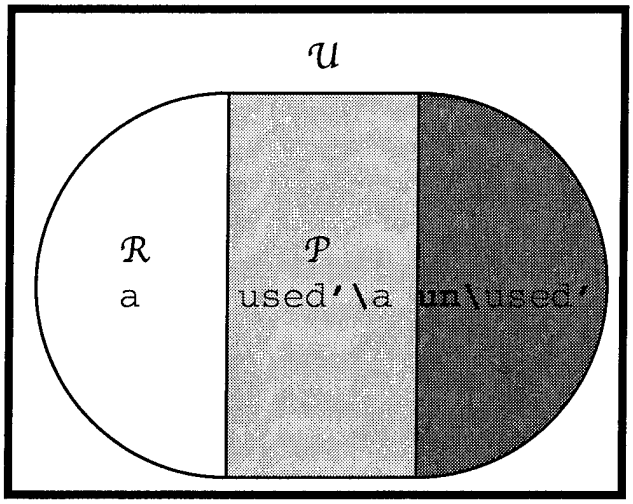

Figure 4: Partitioning $\mathcal{U}$ into three sets for bounded generation of used. The first (leftmost) partition includes all elements that are required to be included in any the value of used for any satisfying assignment. In this example, this is simply the value denoted by the variable a. The term $\mathcal{R}$ is used to describe these values. The second partition includes all elements that may possibly be included in the value of used. The term $\mathcal{P}$ is used to describe these values. The third, and final, partition includes all elements that must not be included in the value of used. In this example, this is the value $\bar{s}_{set}$(Un\ used'). Only the elements in the middle partition need to be considered by the underlying generator.

the sets of elements used for scalar and set generation. This definition, however, significantly complicates both the definition and implementation of bounded generation. Instead, I use sets of elements for bounded generation of relations as well. One set describes the elements that may or may not be in the domain of the relation for a satisfying assignment. I call the term describing this set $\mathcal{P}$, as with scalar or set generation. There is a similar term, which I call $\mathcal{p}$, that describes the set of elements that may or may not be part of the range. As with scalar generation, $\mathcal{R}$ is uninteresting, so $\mathcal{R}$ is always {}[5].

3.2 $\mathcal{P}\mathcal{p}$ Limitation

Bounded generation depends on generating values from a limited universe of elements. In this section, I develop a notation for describing the underlying limitations.

These limitations are based on two elements from Term_{set}, referred to as $\mathcal{P}$ and $\mathcal{p}$. The currently generated partial assignment s provides the context to evaluate these terms, using the $\bar{s}_{set}$ function. For scalar values and set values, only $\mathcal{P}$ is interesting, with $\bar{s}_{set}(\mathcal{P})$ yielding the base set from which elements may be drawn. For relational values, $\bar{s}_{set}(\mathcal{P})$ limits the domain and $\bar{s}_{set}(\mathcal{p})$ limits the range.

Definition 32: For any $\mathcal{P},\mathcal{p} \in \mathrm{Term}_{set}$, value x is $\mathcal{P}\mathcal{p}$-limited for assignment s iff

$$FV(\mathcal{P}) \subseteq \mathrm{dom}\ s \wedge FV(\mathcal{p}) \subseteq \mathrm{dom}\ s \wedge$$
$$(x \in \mathrm{Value}_{scalar} \Rightarrow x \in \bar{s}(\mathcal{P})) \wedge$$

5. As will be shown in Section 6, where I refine the model to distinguish functions from general relations, a term describing a set of edges equivalent to $\mathcal{R}$ is useful for the generation of functions.

$(x \in \text{Value}_{set} \Rightarrow x \subseteq \bar{s}(\mathcal{P})) \wedge$
$(x \in \text{Value}_{rel} \Rightarrow x \subseteq \{ (y,z) \mid y \in \bar{s}(\mathcal{P}) \wedge z \in \bar{s}(p) \})$

The limit functions generate $\mathcal{P}p$-limited values and assignments for most values and assignments. These functions, however, cannot $\mathcal{P}p$-limit a scalar value that is not already $\mathcal{P}p$-limited, or an assignment that maps a variable to a scalar value that is not already $\mathcal{P}p$-limited.

Definition 33: For the function $\text{limit}^{\mathcal{P}p}$:Value×S→Value ∪ {unknown},
$\text{limit}^{\mathcal{P}p}(x,s)$ is
if $\neg (\text{FV}(\mathcal{P}) \subseteq \text{dom } s \wedge \text{FV}(p) \subseteq \text{dom } s)$
unknown
if $x \in \text{Value}_{scalar}$
x if $x \in \bar{s}(\mathcal{P})$
unknown otherwise
if $x \in \text{Value}_{set}$
$\{ y \mid y \in \bar{s}(\mathcal{P}) \wedge y \in x \}$
if $x \in \text{Value}_{rel}$
$\{ (y,z) \mid y \in \bar{s}(\mathcal{P}) \wedge z \in \bar{s}(p) \wedge (y,z) \in x \}$

Definition 34: The function $\text{limit}_i^{\mathcal{P}p}$:S→S is defined as
$\text{limit}_i^{\mathcal{P}p}(s) =$
$\{ v \mapsto x \mid v \in \text{dom } s \wedge \text{Ord}(v) \neq i \wedge x = s(v) \} \cup$
$\{ v \mapsto x \mid v \in \text{dom } s \wedge \text{Ord}(v) = i \wedge x = \text{limit}^{\mathcal{P}p}(s(v),s) \}$

For scalar variables, the function $\text{limit}_i^{\mathcal{P}p}$ leaves v_i unbound in the resultant assignment if the value $s(v)$ is not already $\mathcal{P}p$-limited.

A generator can be $\mathcal{P}p$-limiting.

Definition 35: For any $\mathcal{P},p \in \text{Term}_{set}$ such that $\text{FV}(\mathcal{P}) \subseteq \text{Var}_{i-1}$ and $\text{FV}(p) \subseteq \text{Var}_{i-1}$, a level i generator $g_i^{\mathcal{P}p}$ is a $\mathcal{P}p$-limiting generator iff $\forall s \in S_{i-1}.\forall s' \in g_i^{\mathcal{P}p}(s).s'(v_i)$ is $\mathcal{P}p$-limited for s.

An exhaustive-enumeration $\mathcal{P}p$-limiting generator can be defined.

Definition 36: For some $\mathcal{P},p \in \text{Term}_{set}$ such that $\text{FV}(\mathcal{P}) \subseteq \text{Var}_{i-1}$ and $\text{FV}(p) \subseteq \text{Var}_{i-1}$,
the level i generator $g0^{\mathcal{P}p}$: $S_{i-1} \rightarrow \mathbb{P}S_i$ is defined as
if $v_i \in \text{Var}_{scalar}$
$g0^{\mathcal{P}p}(s) = \{ s' \mid \exists x \in \bar{s}(\mathcal{P}).\ s' = s \cup \{ v_i \mapsto x \} \}$
if $v_i \in \text{Var}_{set}$
$g0^{\mathcal{P}p}(s) = \{ s' \mid \exists x \in \mathbb{P}\bar{s}(\mathcal{P}).\ s' = s \cup \{ v_i \mapsto x \} \}$
if $v_i \in \text{Var}_{rel}$
$g0^{\mathcal{P}p}(s) = \{ s' \mid \exists x \in \mathbb{P}\bar{s}(\mathcal{P}) \times \bar{s}(p).s' = s \cup \{ v_i \mapsto x \} \}$

Theorem 12: The level i generator $g0^{\mathcal{P}p}$ given by Definition 36 is a $\mathcal{P}p$-limiting generator.

Proof: Let $\mathcal{P},p \in \text{Term}_{set}$ such that $\text{FV}(\mathcal{P}) \subseteq \text{Var}_{i-1}$ and $\text{FV}(p) \subseteq \text{Var}_{i-1}$ and $s \in S_{i-1}$.

if $v_i \in Var_{scalar}$

$\forall s' \in g0^{\mathcal{P}p}(s).s'(v_i) \in \bar{s}(\mathcal{P})$

Therefore, $s'(v_i)$ is $\mathcal{P}p$-limited for s.

The other cases follow similarly. ■

Just as a set of assignments can represent a duplication, a set of assignments can be a $\mathcal{P}p$-limited representative of a duplication.

Definition 37: For any $\mathcal{P},p \in Term_{set}$ such that $FV(\mathcal{P}) \subseteq Var_{i-1}$ and $FV(p) \subseteq Var_{i-1}$, a set $L_i \subseteq S_i$ is a $\mathcal{P}p$-limited representative of $d(\phi)$ iff

$\forall s \in S_N. \models\phi[s] = true \Rightarrow \exists s' \in S_N.s \approx_{d(\phi)} s' \wedge limit_i^{\mathcal{P}p}(Var_i \lhd s') \in L_i$.

This is almost the same definition as was used for represents originally. Instead of matching a partial assignment from each equivalence class in $d(\phi)$, it is now necessary to match a $\mathcal{P}p$-limited partial assignment from each equivalence class.

Definition 38: A $\mathcal{P}p$-limited level i generator $g_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for $d(\phi)$ iff

$\forall s \in S_N.\models\phi[s] = true \Rightarrow \exists s' \in S_N.s \approx_{d(\phi)} s' \wedge limit_i^{\mathcal{P}p}(Var_i \lhd s') \in g_i^{\mathcal{P}p}(Var_{i-1} \lhd s)$.

As before, a generator is sound if it yields a value equivalent to each possible value, except it now yields the $\mathcal{P}p$-limited equivalent value. The aggregate generator for a $\mathcal{P}p$-limited sound generator yields a $\mathcal{P}p$-limited representative set when given a $\mathcal{P}p$-limited representative set.

Theorem 13: If a $\mathcal{P}p$-limited level i generator $g_i^{\mathcal{P}p}$ with the aggregate generator $G_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for $d(\phi)$ then

$\forall Q_{i-1} \subseteq S_{i-1}. Q_{i-1}$ represents $d(\phi) \Rightarrow G_i^{\mathcal{P}p}(Q_{i-1})$ $\mathcal{P}p$-limited represents $d(\phi)$.

Proof: By construction.

Assume $s \in S_N.\models\phi[s] = true$.

Because Q_{i-1} represents $d(\phi)$,

$\exists s' \in S_N.s \approx_{d(\phi)} s' \wedge Var_{i-1} \lhd s' \in Q_{i-1}$.

Therefore, by definition of $\mathcal{P}p$-limited sound,

$\exists s'' \in S_N.limit_i^{\mathcal{P}p}(Var_i \lhd s'') \in g_i^{\mathcal{P}p}(Var_{i-1} \lhd s') \wedge s' \approx_{d(\phi)} s''$.

By transitivity, $s \approx_{d(\phi)} s''$.

Therefore, because $limit_i^{\mathcal{P}p}(Var_i \lhd s'') \in G_i^{\mathcal{P}p}(Q_{i-1})$,

$G_i^{\mathcal{P}p}(Q_{i-1})$ is a $\mathcal{P}p$-limited representative of $d(\phi)$. ■

The $\mathcal{P}p$-limited exhaustive-enumeration generator defined earlier is $\mathcal{P}p$-limited sound.

Theorem 14: The exhaustive-enumeration $\mathcal{P}p$-limiting generator $g0^{\mathcal{P}p}$ defined by Definition 36 is a $\mathcal{P}p$-limited sound level i generator for any duplication $d(\phi)$ and formula ϕ such that $v_i \in Var_{scalar} \Rightarrow \phi\models v_i$ in $\mathcal{P}$.

Proof: There are three cases to consider based on the kind of variable

If $v_i \in Var_{scalar}$

By definition, $\models\phi[s] = true \Rightarrow s(v_i) \in \bar{s}(\mathcal{P})$.

By definition of $g0^{\mathcal{P}p}$,

$\forall s \in S_N.\forall x \in \bar{s}(\mathcal{P}).\exists s' \in g0^{\mathcal{P}p}(Var_{i-1} \lhd s).s'(v_i) = x$

Therefore, $\forall s \in S_N.\models\phi[s] = true \Rightarrow s \in g0^{\mathcal{P}p}(Var_{i-1} \lhd s)$

Therefore, $limit_i^{\mathcal{P}p}(Var_i \lhd s) \in g0^{\mathcal{P}p}(Var_{i-1} \lhd s)$

Because $s \approx_{d(\phi)} s$ by definition, $g0^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound.

If $v_i \in Var_{set}$

By definition of $limit^{Pp}$, $\forall s \in S_N.\forall x \in Value_{set}.limit^{\mathcal{P}p}(x,s) \subseteq \bar{s}(\mathcal{P})$.

Therefore, $\forall s \in S_N.limit^{\mathcal{P}p}(s(v_i),s) \subseteq \bar{s}(\mathcal{P})$.

By definition of $g0^{\mathcal{P}p}$, $\forall s \in S_N.\forall x \subseteq \bar{s}(\mathcal{P}).\exists s' \in g0^{\mathcal{P}p}(s).s'(v_i) = x$

Therefore, $\forall s \in S_N.limit_i^{\mathcal{P}p}(Var_i \lhd s) \in g0^{\mathcal{P}p}(s)$.

A similar argument follows when $v_i \in Var_{rel}$. ■

3.3 Definition of Bounded Generation

I now define bounded generation in terms of $\mathcal{P}p$-limiting.

Definition 39: For any $\mathcal{P},p,\mathcal{R} \in Term_{set}$, a level i generator $bg_i^{\mathcal{P}p\mathcal{R}}$ is a bounded generator for ϕ using a $\mathcal{P}p$-limiting generator $g_i^{\mathcal{P}p}$, iff

$FV(\mathcal{P}) \subseteq Var_{i-1} \wedge FV(p) \subseteq Var_{i-1} \wedge FV(\mathcal{R}) \subseteq Var_{i-1} \wedge$

$(v_i \in Var_{scalar} \Rightarrow$

$\phi \models v_i$ in $\mathcal{P} \wedge p = \{\} \wedge \mathcal{R} = \{\} \wedge$

$bg_i^{\mathcal{P}p\mathcal{R}}(s) = g_i^{\mathcal{P}p}(s)) \wedge$

$(v_i \in Var_{set} \Rightarrow$

$\phi \models v_i <= (\mathcal{P} \cup \mathcal{R}) \wedge \phi \models \mathcal{R} <= v_i \wedge p = \{\} \wedge$

$bg_i^{\mathcal{P}p\mathcal{R}}(s) = \{ s' \mid \exists s'' \in g_i^{\mathcal{P}p}(s).\ s' = s'' \cup \bar{s}(\mathcal{R})\}) \wedge$

$(v_i \in Var_{rel} \Rightarrow$

$\phi \models dom\ v_i <= \mathcal{P} \wedge \phi \models ran\ v_i <= p \wedge \mathcal{R} = \{\} \wedge$

$bg_i^{\mathcal{P}p\mathcal{R}}(s) = g_i^{\mathcal{P}p}(s))$

This definition exactly corresponds to the overview of bounded generation given earlier. All the terms used must be well defined for any partial assignment considered as an input. For scalar variables, the generator utilizes an underlying generator that considers only the possible values, as given by the term $\mathcal{P}$. For set variables, the generator unions the required elements, as indicated by the $\mathcal{R}$ term, with each value yielded by the underlying generator considering only the non-required possible elements, as given by the $\mathcal{P}$ term. For relational variables, the generator yields values given by the underlying generator considering a reduced set of possible elements for the domain (indicated by the term $\mathcal{P}$) and for the range (indicated by the term p).

A bounded-generation generator, when combined with a $\mathcal{P}p$-limited sound generator, is sound for appropriate partial assignment duplications.

Theorem 15: If $v_i \in Var_{scalar}$ and $g_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for $PAd(\phi, v_i \text{ in } \mathcal{P})$, then the level i bounded generator $bg_i^{\mathcal{P}p\mathcal{R}}$ using the exhaustive-enumeration $\mathcal{P}p$-limiting generator $g_i^{\mathcal{P}p}$ is sound for $PAd(\phi, v_i \text{ in } \mathcal{P})$.

Proof:

Because $\phi \models v_i \text{ in } \mathcal{P}$, $\forall s \in S_N. \models\phi[s] = true \Rightarrow s(v_i) \in \bar{s}(\mathcal{P})$

By definition, $s(v_i) \in \bar{s}(\mathcal{P}) \Rightarrow limit_i^{\mathcal{P}p}(s(v_i)) = s(v_i)$

Therefore, $\forall s \in S_N. \models\phi[s] = true \Rightarrow limit_i^{\mathcal{P}p}(Var_i \lhd s) = Var_i \lhd s$.

Therefore, $g_i^{\mathcal{P}p}$ is sound for $PAd(\phi, v_i \text{ in } \mathcal{P})$.

Because $bg_i^{\mathcal{P}pR} = g_i^{\mathcal{P}p}$ when $v_i \in Var_{scalar}$,

$bg_i^{\mathcal{P}p\mathcal{R}}$ is sound for $PAd(\phi, v_i \text{ in } \mathcal{P})$. ■

Theorem 16: If $v_i \in Var_{set}$ and $g_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for $PAd(\phi, (v_i <= (\mathcal{P} \cup \mathcal{R}) \text{ and } \mathcal{R} <= v_i))$, then the level i bounded generator $bg_i^{\mathcal{P}p\mathcal{R}}$ using the exhaustive-enumeration $\mathcal{P}p$-limiting generator $g_i^{\mathcal{P}p}$ is sound for $PAd(\phi, (v_i <= (\mathcal{P} \cup \mathcal{R}) \text{ and } \mathcal{R} <= v_i))$.

Proof: By construction.

Let $s \in S_N. \models\phi[s] = true$.

Because $g_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for the duplication,

$\exists s' \in S_N. s \approx_{PAd} s' \wedge limit_i^{\mathcal{P}p}(Var_i \lhd s') \in g_i^{\mathcal{P}p}(Var_{i-1} \lhd s)$.

By the definition of $\approx_{PAd}$ and because $\models\phi[s] = true$, $s \approx_{PAd} s' \Rightarrow s = s'$.

Therefore, $limit_i^{\mathcal{P}p}(Var_i \lhd s) \in g_i^{\mathcal{P}p}(Var_{i-1} \lhd s)$.

By definition, $limit_i^{\mathcal{P}p}(Var_i \lhd s)(v_i) = s(v_i) \cap \bar{s}(\mathcal{P})$.

Because $\phi \models (v_i <= (\mathcal{P} \cup \mathcal{R}) \text{ and } \mathcal{R} <= v_i)$,

$\phi \models v_i <= (\mathcal{P} \cup \mathcal{R}) \wedge \phi \models \mathcal{R} <= v_i$

Therefore, $\models\phi[s] = true \Rightarrow s(v_i) \subseteq \bar{s}((\mathcal{P} \cup \mathcal{R})) \wedge \bar{s}(\mathcal{R}) \subseteq s(v_i)$.

Therefore $s(v_i) \subseteq \bar{s}(\mathcal{P}) \cup \bar{s}(\mathcal{R})$.

Therefore, $s(v_i) \subseteq (s(v_i) \cap \bar{s}(\mathcal{P})) \cup \bar{s}(\mathcal{R})$.

Because $\bar{s}(\mathcal{R}) \subseteq s(v_i)$, $(s(v_i) \cap \bar{s}(\mathcal{P})) \cup \bar{s}(\mathcal{R}) \subseteq s(v_i)$.

Therefore, $(s(v_i) \cap \bar{s}(\mathcal{P})) \cup \bar{s}(\mathcal{R}) = s(v_i)$.

Therefore, $limit_i^{\mathcal{P}p}(Var_i \lhd s)(v_i) \cup \bar{s}(\mathcal{R}) = s(v_i)$.

Therefore $s \in bg_i^{\mathcal{P}p\mathcal{R}}(Var_{i-1} \lhd s)$. ■

Theorem 17: If $v_i \in Var_{rel}$ and $g_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for $PAd(\phi, (\text{dom } v_i <= \mathcal{P} \text{ and ran } v_i <= \mathcal{P}))$, then the level i bounded generator $bg_i^{\mathcal{P}p\mathcal{R}}$ using the exhaustive-enumeration $\mathcal{P}p$-limit-

ing generator $g_i^{\mathcal{P}p}$ is sound for $\mathrm{PAd}(\phi,(\mathrm{dom}\ v_i <= \mathcal{P}\ \mathrm{and}\ \mathrm{ran}\ v_i <= p))$.

Proof: By a similar argument used in the proof for Theorem 16.

4. Isomorph Elimination

This section begins by providing an overview of isomorph elimination. The second subsection defines permutation duplicates based on automorphisms on $\mathcal{U}$, the universe of atomic elements. Using the definition of the formula language, I show that applying an automorphism function to all of the values in the range of an assignment does not change the interpretation of the formula for the modified assignment. I define isomorph elimination in terms of these automorphisms. From this, I show that isomorph-eliminating generators are sound.

In the final subsection , I show that an isomorph-eliminating generator can be used as the underlying $\mathcal{P}p$-limiting generator for sound bounded generation.

4.1 Overview of Isomorph Elimination

Conceptually, isomorph elimination is a direct outgrowth of a simple observation: two isomorphic assignments will give the same interpretation to any formula. Two assignments are *isomorphic* if there is a consistent shuffling of the elements in one assignment, called a *relabeling*, that yields the second assignment. As the elements of $\mathcal{U}$ are unstructured, no two elements are distinguishable, except through prior usage. Therefore, a relabeled assignment must give the same interpretation to a formula as the original assignment.

Because of the incremental approach taken in selective enumeration, an isomorph-eliminating generator considers only relabellings that do not effect the partial assignment already computed. For an initial partial assignment, an isomorph-eliminating generator can safely exclude any value for the new variable that is a relabeling of another value that is generated if that relabeling leaves the initial partial assignment unchanged. In this way, the values already generated in earlier levels limit the possible relabellings to consider in this new level.

Ideally, the generator would guarantee that no assignments generated for a given level of the search tree would be isomorphic to each other. However, there are two difficulties in achieving complete isomorph elimination. As the generator considers only a single partial assignment from the previous level at a time, it cannot guarantee that two assignments generated from two different initial partial assignments are not isomorphic. Secondly, perfect recognition of isomorphs is itself non-polynomial, so acceptable performance requires the use of a heuristic.

Basic isomorph elimination is dependent solely on the formula language, rather then on the actual formula being analyzed, as is bounded generation. Although further reductions can be gained by considering the formula [JJD98], I ignore those considerations in this analysis.

4.2 Automorphisms

An automorphism is a function that performs a consistent relabeling of a value or an assignment.

Definition 40: A one-to-one function h:Value$\rightarrow$Value is an automorphism for $\mathcal{U}$ iff

$$\forall y \in \text{Value}_{\text{set}}.\forall x \in \text{Value}_{\text{scalar}}.x \in y \Leftrightarrow h(x) \in h(y) \wedge$$
$$\forall z \in \text{Value}_{\text{rel}}.\forall x,x' \in \text{Value}_{\text{scalar}}.x' \in z(x) \Leftrightarrow h(x') \in h(z)(h(x))$$

Definition 41: For any automorphism h, $h(s) = \{ v \mapsto x \mid v \in \text{dom } s \wedge x = h(s(v)) \}$.
Furthermore, $h(\bar{s})$ is the semantic function induced by $h(s)$.

Because all of the elements of $\mathcal{U}$ are unstructured, applying any automorphism to an assignment does not change the meaning of that assignment for any terms or formulae.

Lemma 18: For any automorphism h, term τ, and assignment s,
$\bar{s}(\tau) \neq \text{unknown} \Rightarrow h(\bar{s}(\tau)) = h(\bar{s})(\tau)$.

Proof: By structural induction on the definition of $\bar{s}(\tau)$.

If τ is v where $v \in \text{Variable}$
By definition of $\bar{s}$, $\bar{s}(\tau) = s(v)$.
By definition of $h(\bar{s})$, $h(\bar{s})(\tau) = h(s(v))$.

if τ is $(\tau_1 \cup \tau_2)$ where $\tau_1, \tau_2 \in \text{Term}_{\text{set}}$
Because $\bar{s}(\tau) \neq \text{unknown}$, $\bar{s}(\tau_1) \neq \text{unknown}$ and $\bar{s}(\tau_2) \neq \text{unknown}$.
By induction, $h(\bar{s}(\tau_1)) = h(\bar{s})(\tau_1)$ and $h(\bar{s}(\tau_2)) = h(\bar{s})(\tau_2)$.
Therefore, by definition of $\bar{s}$, $h(\bar{s}(\tau)) = h(\bar{s})(\tau)$

Other productions follow similarly ■

Theorem 19: For any automorphism h, formula ϕ, and assignment s,
$\models\phi[s] \neq \text{unknown} \Rightarrow \models\phi[s] = \models\phi[h(s)]$.

Proof: By Lemma 18 and structural induction on the definition of the formula language.

If ϕ is τ_1 in τ_2 where $\tau_1 \in \text{Term}_{\text{scalar}}$ and $\tau_2 \in \text{Term}_{\text{set}}$
Because $\models\phi[s] \neq \text{unknown}$, $\bar{s}(\tau_1) \neq \text{unknown}$ and $\bar{s}(\tau_2) \neq \text{unknown}$.
By definition of $\models$, if $\models\phi[s] = \text{true}$, $\bar{s}(\tau_1) \in \bar{s}(\tau_2)$.
Therefore, by Definition 41, $h(\bar{s}(\tau_1)) \in h(\bar{s}(\tau_2))$.
Therefore, by Lemma 18, $h(\bar{s})(\tau_1) \in h(\bar{s})(\tau_2)$.
Therefore, $\models\phi[h(s)] = \text{true}$
By definition of $\models$, if $\models\phi[s] = \text{false}$, $\bar{s}(\tau_1) \notin \bar{s}(\tau_2)$.
Therefore, by Definition 41, $h(\bar{s}(\tau_1)) \notin h(\bar{s}(\tau_2))$.
Therefore, by Lemma 18, $h(\bar{s})(\tau_1) \notin h(\bar{s})(\tau_2)$.
Therefore, $\models\phi[h(s)] = \text{false}$

If ϕ is $(\phi_1$ and $\phi_2)$ where $\phi_1, \phi_2 \in \text{Wff}$
If $\models\phi_1[s] = \text{true}$, $\models\phi_1[h(s)] = \text{true}$ by induction
If $\models\phi_2[s] = \text{true}$, $\models\phi_2[h(s)] = \text{true}$ by induction
Therefore, by definition of $\models$, if $\models\phi[s] = \text{true}$, $\models\phi[h(s)] = \text{true}$
If $\models\phi_1[s] = \text{false}$, $\models\phi_1[h(s)] = \text{false}$ by induction
If $\models\phi_1[s] = \text{false}$, $\models\phi_1[h(s)] = \text{false}$ by induction
Therefore, by definition of $\models$, if $\models\phi[s] = \text{false}$, $\models\phi[h(s)] = \text{false}$

Other productions follow similarly ■

An isomorph-eliminating generator considers only automorphisms that leave the initial partial assignment unchanged. These automorphisms are called identities.

Definition 42: An automorphism $\hat{h}$ is an identity for an assignment s iff $s = \hat{h}(s)$

For any empty assignment or an assignment containing only empty sets and empty relations, any automorphism is an identity. For the assignment considered earlier

$$\{ \text{usage}' \mapsto \{ (a_0, v_0) \}, \text{used}' \mapsto \varnothing \}$$

any automorphism that maps a_0 to a_0 and v_0 to v_0 is an identity.

Identities for the more complicated assignment

$$\{ \text{usage}' \mapsto \{ (a_0, v_0), (a_1, v_0) \}, \text{used}' \mapsto \{a_0, a_1 \} \}$$

include automorphisms that map a_0 to a_1, a_1 to a_0, and v_0 to v_0 as well as the expected automorphisms that map a_0 to a_0, a_1 to a_1, and v_0 to v_0.

If an automorphism is an identity over a partial assignment, it is an identity for any value obtained by evaluating a term using that assignment.

Lemma 20: If $\hat{h}$ is an automorphism that is an identity for a partial assignment $s \in S_i$ and $\tau \in \text{Term}$ such that $FV(\tau) \subseteq \text{Var}_i$, then $\hat{h}(\bar{s}(\tau)) = \bar{s}(\tau)$.

Proof: By structural induction on term language.

By Theorem 1, $\bar{s}(\tau) \neq \text{unknown}$.

If τ is v, where $v \in \text{Variable}$

$\bar{s}(\tau) = s(\tau)$ by definition

Because $\hat{h}$ is an identity for s, $\hat{h}(s(\tau)) = s(\tau)$

If τ is $(\tau_1 \cup \tau_2)$, where $\tau_1, \tau_2 \in \text{Term}_{set}$

By induction, $\hat{h}(\bar{s}(\tau_1)) = \bar{s}(\tau_1)$ and $\hat{h}(\bar{s}(\tau_2)) = \bar{s}(\tau_2)$

Therefore, $\hat{h}(s(\tau)) = s(\tau)$

Other productions follow similarly. ■

4.3 Definition of Isomorph Elimination

To define isomorph elimination precisely, it is useful to start with a definition of the duplication being reduced by the generator. The duplication places any two assignments in the same equivalence class if they are isomorphic to each other.

Definition 43: An equivalence relation $\approx_\pi$ is a permutation equivalence relation iff

$\forall s, s' \in S_N.\ s \approx_\pi s' \Leftrightarrow (\exists \hat{h}.\ \hat{h}$ is an automorphism function and $s' = \hat{h}(s))$

Lemma 21: The partitioning defined by $\approx_\pi$ is a duplication.

Proof: By Theorem 19.

Definition 44: $\pi d(\phi)$ is the duplication induced by $\approx_\pi$.

The goal of an isomorph-eliminating generator is to generate only assignments that are not isomorphs of other assignments generated.

Definition 45: A level i generator g_i is an isomorph-eliminating generator iff

$$\forall s \in S_{i-1}.\forall x \in \text{Value}.\exists s' \in g_i(s).$$
$$(\exists h.h \text{ is an automorphism} \wedge h \text{ is an identity for } s \wedge s'(v_i) = h(x))$$

An isomorph-eliminating generator is any generator that is guaranteed to generate at least any value for this level that is not isomorphic to another value also generated. As was noted earlier, it is not realistic to require an isomorph-eliminating generator to remove all isomorphs. As such, the exhaustive-enumeration generator given in Definition 21 is a valid isomorph-eliminating generator, albeit an extraordinarily inefficient one. In practice, a middle ground is available; an efficient generator that eliminates almost all isomorphs can be implemented with reasonable effort.

Because an isomorph-eliminating generator only drops an assignment if there is an automorphism that relabels that assignment into one that is generated, there is at least one element generated for each equivalence class in $\pi d(\phi)$.

Theorem 22: An isomorph-eliminating generator g_i is sound for $\pi d(\phi)$.

Proof: Assume $s \in S_N.\models\phi[s] = \text{true}$.

Let $x = s(v_i)$.

By definition of an isomorph-eliminating generator,

$$\exists s' \in g_i(\text{Var}_{i-1} \lhd s).$$
$$(\exists h.h \text{ is an automorphism} \wedge$$
$$h \text{ is an identity for } s \wedge s'(v_i) = h(x))$$

Therefore, $h(s) = s'$.

Therefore, $s \approx_\pi s'$. ■

4.4 Interaction of Isomorph Elimination and Bounded Generation

To minimize the number of assignments generated, selective enumeration utilizes all the duplications available. As shown in Theorem 6, different partial assignment duplications combine to reduce the number of assignments for a single formula in a straightforward manner. The complexity comes when combining a partial assignment duplication with a permutation duplication: $PAd(\phi,\phi') \circ \pi d(\phi)$.

If the partial assignment duplication enables a derived variable, only one assignment will be generated and combination with isomorph elimination is unnecessary. Short circuiting, as described in the next section, is really more of a post-filter than a generator and combines with any other generators in a straightforward manner. The issue is therefore limited to combining isomorph-eliminating generators with bounded-generation generators. The approach is to utilize an isomorph-eliminating generator as the underlying generator for a bounded-generation generator.

This requires a $\mathcal{P}p$-limiting version of an isomorph-eliminating generator to be defined.

Definition 46: For $\mathcal{P},p \in \text{Term}_{set}$ such that $FV(\mathcal{P}) \subseteq \text{Var}_{i-1}$ and $FV(p) \subseteq \text{Var}_{i-1}$, a level i generator $g_i^{\mathcal{P}p}$ is a $\mathcal{P}p$-isomorph-eliminating generator for formula ϕ iff

$$
\begin{aligned}
v_i \in Var_{scalar} \Rightarrow\ & \forall s \in S_{i-1}.\forall x \in Value_{scalar}. \\
& (x \notin \bar{s}(\mathcal{P})) \vee \\
& \exists s' \in g_i(s). \\
& \quad (\exists h.h \text{ is an automorphism} \wedge \\
& \quad\quad h \text{ is an identity for } s \wedge \\
& \quad\quad s'(v_i) = h(\text{limit}^{\mathcal{P}p}(x,s))) \wedge \\
v_i \in Var_{set} \Rightarrow\ & \forall s \in S_{i-1}.\forall x \in Value_{set}. \\
& \exists s' \in g_i(s). \\
& \quad (\exists h.h \text{ is an automorphism} \wedge \\
& \quad\quad h \text{ is an identity for } s \wedge \\
& \quad\quad s'(v_i) = h(\text{limit}^{\mathcal{P}p}(x,s))) \wedge. \\
v_i \in Var_{rel} \Rightarrow\ & \forall s \in S_{i-1}.\forall x \in Value_{rel}. \\
& \exists s' \in g_i(s). \\
& \quad (\exists h.h \text{ is an automorphism} \wedge \\
& \quad\quad h \text{ is an identity for } s \wedge \\
& \quad\quad s'(v_i) = h(\text{limit}^{\mathcal{P}p}(x,s))).
\end{aligned}
$$

A $\mathcal{P}p$-isomorph-eliminating generator generates a $\mathcal{P}p$-limited value for each possible value or it generates a relabelling of the $\mathcal{P}p$-limited value. But a relabelling of the $\mathcal{P}p$-limited value is itself a $\mathcal{P}p$-limited value. Therefore, a $\mathcal{P}p$-isomorph-eliminating generator is a $\mathcal{P}p$-limiting generator.

Lemma 23: A $\mathcal{P}p$-isomorph-eliminating generator is a $\mathcal{P}p$-limiting generator.

Proof: Need to prove that $\forall s \in S_{i-1}.\forall s' \in g_i^{\mathcal{P}p}(s).s'(v_i)$ is $\mathcal{P}p$-limited.

By definition, $\exists x.s'(v_i) = h(\text{limit}^{\mathcal{P}p}(x,s))$.

Because h is an identity for s and $FV(\mathcal{P}) \subseteq \text{dom } s$ and $FV(p) \subseteq \text{dom } s$,
$h(\bar{s}(\mathcal{P})) = \bar{s}(\mathcal{P})$ and $h(\bar{s}(p)) = \bar{s}(p)$.

If $v_i \in Var_{scalar}$,
$\text{limit}^{\mathcal{P}p}(x,s) = x$
By definition, $x \in \bar{s}(\mathcal{P}) \Rightarrow h(x) \in h(\bar{s}(\mathcal{P}))$
Because $h(\bar{s}(\mathcal{P})) = \bar{s}(\mathcal{P})$, $h(x) \in \bar{s}(\mathcal{P})$
Therefore, $s'(v_i)$ is $\mathcal{P}p$-limited

If $v_i \in Var_{set}$
$\text{limit}^{\mathcal{P}p}(x,s) = x \cap \bar{s}(\mathcal{P})$
so $h(\text{limit}^{\mathcal{P}p}(x,s)) = h(x) \cap h(\bar{s}(\mathcal{P}))$
so $h(\text{limit}^{\mathcal{P}p}(x,s)) = h(x) \cap \bar{s}(\mathcal{P})$
Therefore, $s'(v_i)$ is $\mathcal{P}p$-limited

If $v_i \in Var_{rel}$
$\text{limit}^{\mathcal{P}p}(x,s) = x \cap \bar{s}(\mathcal{P}) \times \bar{s}(p)$
so $h(\text{limit}^{\mathcal{P}p}(x,s)) = h(x) \cap h(\bar{s}(\mathcal{P})) \times h(\bar{s}(p))$
so $h(\text{limit}^{\mathcal{P}p}(x,s)) = h(x) \cap \bar{s}(\mathcal{P}) \times \bar{s}(p)$
Therefore, $s'(v_i)$ is $\mathcal{P}p$-limited

Therefore, $g_i^{\mathcal{P}p}$ is a $\mathcal{P}p$-limiting generator. ■

A $\mathcal{P}p$-isomorph-eliminating generator is also $\mathcal{P}p$-limited sound.

Lemma 24: A $\mathcal{P}p$-isomorph-eliminating generator $g_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for $\pi d(\phi)$.

Proof: Assume $s \in S_N.\models\phi[s] = true$.

Need to find an element $s' \in S_N.s \approx_{\pi d(\phi)} s' \wedge Var_i \lhd s' \in g_i^{\mathcal{P}p}(Var_{i-1} \lhd s)$

By definition, $\exists s'' \in g_i^{\mathcal{P}p}(Var_{i-1} \lhd s)$.
$\exists h.h(s(v_i)) = s''(v_i) \wedge h$ is an identity for $Var_{i-1} \lhd s$.

Therefore, $h((Var_i \lhd s) = s''$.

Therefore $\exists s' \in S_N.s \approx_{\pi d(\phi)} s' \wedge Var_i \lhd s' = s''$.

Therefore, $Var_i \lhd s' \in g_i^{\mathcal{P}p}(Var_{i-1} \lhd s)$. ■

Together, a $\mathcal{P}p$-isomorph-eliminating generator and a bounded-generation generator generate fewer assignments than either a simple isomorph-eliminating generator or a bounded-generation generator paired with an exhaustive-enumeration generation. The combination, however, is still sound, now for the combination of an appropriate partial assignment duplication with the permutation duplication.

Theorem 25: If $v_i \in Var_{scalar}$ and $g_i^{\mathcal{P}p}$ is a $\mathcal{P}p$-limiting isomorph-eliminating generator, then the level i bounded generator $bg_i^{\mathcal{P}p\mathcal{R}}$ using $g_i^{\mathcal{P}p}$ is sound for $PAd(\phi, v_i \text{ in } \mathcal{P}) \circ \pi d(\phi)$.

Proof: Assume $s \in S_N$ such that $\models\phi[s] = true$.

Need to prove $\exists s' \in S_N.s \approx s' \wedge Var_i \lhd s' \in bg_i^{\mathcal{P}p\mathcal{R}}(s)$.

Because $\phi \models v_i$ in $\mathcal{P}$, $\forall s \in S_N.\ \models\phi[s] = true \Rightarrow s(v_i) \in \bar{s}(\mathcal{P})$
By definition, $s(v_i) \in \bar{s}(\mathcal{P}) \Rightarrow limit_i^{\mathcal{P}p}(s(v_i)) = s(v_i)$
Therefore, $\forall s \in S_N.\ \models\phi[s] = true \Rightarrow limit_i^{\mathcal{P}p}(Var_i \lhd s) = Var_i \lhd s$.

By Lemma 24, $g_i^{\mathcal{P}p}$ is $\mathcal{P}p$-limited sound for $\pi d(\phi)$.

Therefore, $g_i^{\mathcal{P}p}$ is sound for $\pi d(\phi)$.

Therefore, by Theorem 9, $g_i^{\mathcal{P}p}$ is sound for $PAd(\phi, v_i \text{ in } \mathcal{P}) \circ \pi d(\phi)$.

Since $bg_i^{\mathcal{P}p\mathcal{R}}$ simply passes through the assignments generated by $g_i^{\mathcal{P}p}$,
$bg_i^{\mathcal{P}p\mathcal{R}}$ is sound for $PAd(\phi, v_i \text{ in } \mathcal{P}) \circ \pi d(\phi)$. ■

Theorem 26: If $v_i \in Var_{set}$ and $g_i^{\mathcal{P}p}$ is a $\mathcal{P}p$-limiting isomorph-eliminating generator, then the level i bounded generator $bg_i^{\mathcal{P}p\mathcal{R}}$ using $g_i^{\mathcal{P}p}$ is sound for $PAd(\phi, (v_i \text{ in } (\mathcal{P} \cup \mathcal{R}) \text{ and } \mathcal{R} <= v_i)) \circ \pi d(\phi)$.

Proof: Assume $s \in S_N$ such that $\models\phi[s] = true$.

By definition of $g_i^{\mathcal{P}p}$,
$\exists s' \in g_i(s)$.
($\exists h.h$ is an automorphism $\wedge$
h is an identity for $s \wedge$

$$s'(v_i) = \hbar(\text{limit}^{\mathcal{P}p}(s(v_i),s))).$$

Therefore, $\hbar(\text{limit}_i^{\mathcal{P}p}(\text{Var}_i \lhd s)) = s'$.

By definition, $\text{limit}_i^{\mathcal{P}p}(\text{Var}_i \lhd s)(v_i) = s(v_i) \cap \bar{s}(\mathcal{P})$.

Because $\phi \models (v_i <= (\mathcal{P} \cup \mathcal{R})$ and $\mathcal{R} <= v_i)$,

$$\phi \models v_i <= (\mathcal{P} \cup \mathcal{R}) \wedge \phi \models \mathcal{R} <= v_i$$

Therefore, $\models \phi[s] = \text{true} \Rightarrow s(v_i) \subseteq \bar{s}((\mathcal{P} \cup \mathcal{R})) \wedge \bar{s}(\mathcal{R}) \subseteq s(v_i)$.

Therefore $s(v_i) \subseteq \bar{s}(\mathcal{P}) \cup \bar{s}(\mathcal{R})$.

Therefore, $s(v_i) \subseteq (s(v_i) \cap \bar{s}(\mathcal{P})) \cup \bar{s}(\mathcal{R})$.

Because $\bar{s}(\mathcal{R}) \subseteq s(v_i)$, $(s(v_i) \cap \bar{s}(\mathcal{P})) \cup \bar{s}(\mathcal{R}) \subseteq s(v_i)$.

Therefore, $(s(v_i) \cap \bar{s}(\mathcal{P})) \cup \bar{s}(\mathcal{R}) = s(v_i)$.

Therefore, $\text{limit}_i^{\mathcal{P}p}(\text{Var}_i \lhd s)(v_i) \cup \bar{s}(\mathcal{R}) = s(v_i)$.

Therefore, $\text{limit}_i^{\mathcal{P}p}(\text{Var}_i \lhd s) = \text{Var}_i \lhd s$.

Therefore, $\hbar(\text{Var}_i \lhd s) = s'$.

Therefore, $\exists s'' \in S_N.\hbar(s) = s''$.

Therefore, $s \approx s'' \wedge \text{Var}_i \lhd s'' \in g_i(s)$. ■

Theorem 27: If $v_i \in \text{Var}_{\text{rel}}$ and $g_i^{\mathcal{P}p}$ is a $\mathcal{P}p$-limiting isomorph-eliminating generator, then the level i bounded generator $bg_i^{\mathcal{P}p\mathcal{R}}$ using $g_i^{\mathcal{P}p}$ is sound for $\text{PAd}(\phi,(\text{dom } v_i <= \mathcal{P}$ and $\text{ran } v_i <= \mathcal{p})) \circ \pi d(\phi)$.

Proof: By a similar argument to the one used to prove Theorem 26.

5. Derived Variables and Short Circuiting

This section formalizes the selective enumeration techniques called derived variables and short circuiting, which were introduced by example in the first section. Each technique will be proven sound.

These techniques are far simpler than isomorph elimination or bounded generation. These techniques also lack the complications of combining techniques found with bounded generation and isomorph elimination.

5.1 Overview of Derived Variables

A derived variable is one that has a constructive derivation. Typically, one of the atomic formulae conjoined in the formula being analyzed gives an explicit value for the variable. This occurs several places in the example claim uniqueAddrAlloc. In the schema Heap, the atomic formula used = dom usage restricts used to a single value for each value of usage. Similarly, usage could be derived from the variables used and usage', using the atomic formula (used <: usage') = usage. Clearly, the use of one of these derivations invalidates the other; either usage must be generated before used or used must be generated before usage. The derivations available is both an input to and a result of the choice of variable ordering.

Derived variables can be considered as the ultimate application of bounded generation. For derived variables, the value of the ariable is determined by the value of a term that I call $\mathcal{T}$. A scalar variable is a derived variable if the bounded generation term $\mathcal{P}$ is limited to a single element for all possible partial assignments. The term $\mathcal{T}$ defines that single element. Similarly, a set variable is a derived variable if the best bounded generation term $\mathcal{P}$ is empty for all assignments; in this case the derivation term $\mathcal{T}$ is the same as the bounded generation term $\mathcal{R}$. The limitations on bounded generations of relations prohibit a direct equivalence with derived variables, however.

5.2 Definition of Derived-Variable Generation

A variable may be derived if the value is constrained to a single term and that term involves only variables that precede the derived variable in Ord.

Definition 47: For a term $\mathcal{T}$, a level i generator $g_i^{\mathcal{T}}$ is a derived-variable generator for formula ϕ, iff

$$\phi \models v_i = \mathcal{T} \wedge FV(\mathcal{T}) \subseteq Var_{i-1} \wedge \forall s \in S_{i-1} . g_i^{\mathcal{T}}(s) = s \cup \{v_i \mapsto \bar{s}(\mathcal{T})\}$$

A derived-variable generator is always sound for any duplication.

Theorem 28: A level i derived-variable generator $g_i^{\mathcal{T}}$ for formula ϕ is sound for any duplication $d(\phi)$.

Proof: Let $s \in S_N . \models \phi[s] = true$.

Because $\phi \models v_i = \mathcal{T}$, $\Rightarrow s(v_i) = \bar{s}(\mathcal{T})$.

Therefore $Var_i \lhd s \in g_i^{\mathcal{T}}(Var_{i-1} \lhd s)$ ■

5.3 Overview of Short Circuiting

Short circuiting is more of a post-filter than a generator technique. Short circuiting removes all remaining partial assignment duplicates. Given a set of formulae that must be satisfied, short circuiting removes any partial assignments that fail to satisfy one or more of the formulae.

In the example considered earlier, bounded generation had limited the values generated for usage such that the domain of usage was a subset of the domain of usage' and the range of usage was a subset of the range of usage'. The formula, however, includes a stronger constraint: (used <: usage') = usage. Short circuiting removes any partial assignments involving used, usage, and usage' that fail to satisfy this constraint.

Like bounded generation, a short-circuiting generator depends on another generator to produce assignments, modifying the set of assignments generated. Unlike bounded generation, a short-circuiting generator may suppress some assignments generated by the underlying generator, preventing their consideration in the next level of the search.

5.4 Definition of Short Circuiting

A short-circuiting generator requires a set of formulae, referred to as $\mathcal{F}$, and an underlying generator. The short-circuiting generator yields every assignment yielded by the underlying generator that satisfy all of the formulae within $\mathcal{F}$.

Definition 48: For a set of formulae $\mathcal{F}$, a level i generator $g_i^{\mathcal{F}}$ is a short-circuiting generator for formula ϕ using a level i generator g'_i iff
$\forall f \in \mathcal{F}.(\phi \models f \wedge \forall s \in S_{i-1}.g_i^{\mathcal{F}}(s) = \{s' \mid s' \in g'_i(s) \wedge \models f[s'] = true\})$.

A short-circuiting generator is sound if the underlying generator is sound.

Theorem 29: A level i short-circuiting generator $g_i^{\mathcal{F}}$ for formula ϕ using a level i generator g'_i is sound for $PAd(\phi,\phi') \circ d(\phi)$ if g'_i is sound for $d(\phi)$ where ϕ' is f_1 and f_2 and f_3 ...and f_i with $f_1..f_i \in \mathcal{F}$.

Proof: Proof by construction.

Assume $s \in S_N.\models\phi[s] = true$.

Because g'_i is sound for $d(\phi)$, g'_i is sound for $PAd(\phi,\phi') \circ d(\phi)$ by Theorem 9.

Therefore, $\exists s' \in S_N.s \approx s' \wedge Var_i \lhd s \in g'_i(Var_{i-1} \lhd s)$.

Because $\models\phi[s] = true \wedge s \approx s'$, $\models\phi[s'] = true$.

Because $\forall f \in \mathcal{F}.\phi \models f$, $\forall f \in \mathcal{F}.\models f[s'] = true$.

Therefore, $s' \in g_i^{\mathcal{F}}(s)$. ■

6. Refining the Model

The model of selective enumeration developed thus far is somewhat limited. The first subsection here re-introduces given types. Given types are a partitioning of $\mathcal{U}$ that limits the acceptable values for variables. The NP language allows relational variables to be limited to subsets of $\text{Value}_{\text{rel}}$, such as functions, injections, or surjections. Limiting the variable to denote only relations that are functions is the most common of these limitations. The second subsection will focus on this functional limitation.

6.1 Reconsidering Given Types

Given types are disjoint subsets of $\mathcal{U}$, the universe of elements. In NP, a variable or expression is not simply restricted to denoting a scalar, set, or relational value; instead, it is restricted to denoting an element of a given type, or a subset of a given type, or a relation drawn from the cross product of given types.

This information is most easily captured as additional constraints on the formula under consideration. For example, the uniqueAddrAlloc claim is translated and negated into Formula 3; in addition to the requirements given by Formula 2, the first four lines represent the constraint implied by the use of the given types in the variable declarations.

Formula 3: ϕ =
```
( ( ( ( ( ( dom usage <= Addr and ran usage <= Value) and
                    ( dom usage' <= Addr and ran usage' <= Value) ) and
                    ( used' <= Addr and used <= Addr) ) and
                        a in Addr) and
                (dom usage = used and dom usage' = used') ) and
        ( func usage and func usage' ) ) and
        ( ( (used <: usage') = usage and used' = (used U {a}) )
        and a in used) )
```

This new formula uses two new variables, Addr and Value, representing the two given types. These variables are generated before any standard variables using a special generator, called a *scope generator*. The scope generator uses the *scope*, a user-provided mapping from given type to a number of elements desired in each given type.

Definition 49: $\text{Variable}_{\text{all}} = \text{Variable}_{\text{scalar}} \cup \text{Variable}_{\text{set}} \cup \text{Variable}_{\text{rel}} \cup \text{Variable}_{\text{type}}$

Definition 50: A function σ:$\text{Variable}_{\text{type}} \rightarrow 1..N$ is a Scope iff Σ ran $\sigma \leqslant N$

Definition 51: A scope generator sg: $\text{Variable}_{\text{type}} \times \text{Scope} \rightarrow \mathbb{P}\mathcal{U}$ is defined as

$$\text{sg}(t,\sigma) = T.\ |T| = \sigma(t) \wedge x \in T \Rightarrow x \notin \bigcup_{t_i \neq t} \text{sg}(t_i,\sigma)$$

Bounded generation and derived variables can now guarantee that only values satisfying the given type constraints will be generated.

This does have a notable practical effect on isomorph elimination, however. Because the given types will always be generated first, only automorphisms that do not map any elements between given types will be identities over any partial assignment. Therefore the additional reductions

gained by bounded generation includes many reductions lost in isomorph elimination. However, this restriction can also be used to advantage in the implementation of isomorph elimination; a much smaller space of automorphisms need be considered during generation.

6.2 Limiting Relations to Functions

The original specification restricted the values of usage and usage' to be functions, whereas the translated formula allowed these variables to be mapped to any relation. A further improvement on the formula translation is required. Formula 4 adds another constraint involving these variables to the formula given in Formula 3.

Formula 4: ϕ = (((((((dom usage <= Addr and ran usage <= Value) and
(dom usage' <= Addr and ran usage' <= Value)) and
(used' <= Addr and used <= Addr)) and
a in Addr) and
(func usage and func usage')) and
(dom usage = used and dom usage' = used')) and
(func usage and func usage')) and
(((used <: usage') = usage and used' = (used U {a}))
and a in used))

Short circuiting could easily handle these additional constraints directly by including func usage and func usage' as two of the formulae in $\mathcal{F}$. However, this misses some important opportunities to further reduce the number of assignments generated.

The exhaustive-enumeration generator and the isomorph-eliminating generators can be easily enhanced to generate only functions. Bounded generation can also be extended to take advantage of this case without much effort.

Definition 52: For any $\mathcal{P},\mathcal{p},\mathcal{R} \in \mathrm{Term}_{set}$ and $\mathcal{E} \in \mathrm{Term}_{rel}$, a level i generator $bg_i^{\mathcal{P}\mathcal{p}\mathcal{R}\mathcal{E}}$ is a function-aware bounded generator for ϕ using a $\mathcal{P}\mathcal{p}$-limiting generator $g_i^{\mathcal{P}\mathcal{p}}$, iff

$FV(\mathcal{P}) \subseteq Var_{i-1} \wedge FV(\mathcal{p}) \subseteq Var_{i-1} \wedge FV(\mathcal{R}) \subseteq Var_{i-1} \wedge FV(\mathcal{E}) \subseteq Var_{i-1} \wedge$

$(v_i \in Var_{scalar} \Rightarrow$
$\quad \phi \vDash v_i \text{ in } \mathcal{P} \wedge \mathcal{p} = \{\} \wedge \mathcal{R} = \{\} \wedge \mathcal{E} = \{\} \wedge$
$\quad bg_i^{\mathcal{P}\mathcal{p}\mathcal{R}\mathcal{E}}(s) = g_i^{\mathcal{P}\mathcal{p}}(s)) \wedge$

$(v_i \in Var_{set} \Rightarrow$
$\quad \phi \vDash v_i <= (\mathcal{P} \cup \mathcal{R}) \wedge \phi \vDash \mathcal{R} <= v_i \wedge \mathcal{p} = \{\} \wedge \mathcal{E} = \{\} \wedge$
$\quad bg_i^{\mathcal{P}\mathcal{p}\mathcal{R}\mathcal{E}}(s) = \{ s' \mid \exists s'' \in g_i^{\mathcal{P}\mathcal{p}}(s).\ s' = s'' \cup \bar{s}(\mathcal{R}) \}) \wedge$

$(v_i \in Var_{rel} \wedge \phi \vDash \text{func } v_i \Rightarrow$
$\quad \phi \vDash \text{dom } v_i <= (\mathcal{P} \cup \text{dom } \mathcal{E}) \wedge \phi \vDash \text{ran } v_i <= \mathcal{p} \wedge \phi \vDash \mathcal{E} <= v_i \wedge \mathcal{R} = \{\} \wedge$
$\quad bg_i^{\mathcal{P}\mathcal{p}\mathcal{R}\mathcal{E}}(s) = \{ s' \mid \exists s'' \in g_i^{\mathcal{P}\mathcal{p}}(s).\ s' = s'' \cup \bar{s}(\mathcal{E}) \}) \wedge$

$(v_i \in Var_{rel} \wedge \neg\phi \vDash \text{func } v_i \Rightarrow$
$\quad \phi \vDash \text{dom } v_i <= \mathcal{P} \wedge \phi \vDash \text{ran } v_i <= \mathcal{p} \wedge \mathcal{R} = \{\} \wedge \mathcal{E} = \{\} \wedge$
$\quad bg_i^{\mathcal{P}\mathcal{p}\mathcal{R}\mathcal{E}}(s) = g_i^{\mathcal{P}\mathcal{p}}(s))$

This generator works for functions much the way the original generator works for sets. $\mathcal{E}$ is a term describing a set of edges that must be included in any value generated. Because the value must be a function, the domain of $\mathcal{E}$ can be excluded from $\mathcal{P}$.

7. Conclusion

This paper has introduced a formal framework for describing selective enumeration and the techniques that implement it. In related papers, I define specific algorithms for implementing each technique and show that they lead to sound generators. I also define an algorithm for discovering a set of constraining formulae, as needed by partial assignment duplications, and an algorithm for selecting an ordering that takes substantial advantage of the reduction opportunities.

7.1 Future Work

One potential future direction is the search for additional forms of duplication. Although the two duplications I describe in this paper are effective at reducing the number of assignments generated, further duplications would enable additional specifications to be analyzed.

Another possible direction is analyzing different input languages. A promising candidate is OCL[IBM97], the constraint language recently defined for the object specification notation UML. Most of OCL can be directly translated into the formula language I use here. The object orientation introduces some new concepts, particularly inheritance, that require additional consideration and may enable additional constraints.

A final possible direction involves applying the general approach of selective enumeration to incremental search problems in other, non-relational domains. If any easily computable features can be defined that distinguish interesting and non-interesting instances, the framework I described in this paper can be applied.

As an example, consider the problem of TF-sensitive test generation [CM94]. A test T for a boolean expression E is sensitive to a variable x in E if changing only the value of x changes the result of T for E. The goal in test generation is to obtain a near-minimal test set that contains at least one test that is sensitive for each variable. Like the relational satisfaction problem, the general solution is exponential.

Conceptually, a selective-enumeration-like search could be used to find such a set. It seems likely that techniques could be developed that efficiently rule out tests as duplicates if they are not sensitive to any variables not already tested by a test in the set.

8. Bibliography

[[BH94] Rudolf Berghammer and Claudia Hattensperger. *Computer-Aided Manipulation of Relational Expressions and Formulae Using RALF.* Technical Report, Institut fur Informatik und Praktische Mathematik, Christian-Albrechts Universitat Zu Kiel, Kiel, Germany, 1994.

[BJR97] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modelling Language for Object-Oriented Development.* Documentation Set, version 1.0, Rational Software Corporation, Available at <http://www.rational.com>.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys,* Vol. 24, No. 3, September 1992, pp. 293–318.

[BC+92] J. R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 1030 States and Beyond. *Information and Computation,* Vol. 98, No. 2 (June 1992), pp. 143–170.

[Che76] Peter P. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems,* Vol. 1, No. 1, 1976, pp. 9–36.

[CE81] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Logic of Programs: Workshop* (Yorktown Heights, NY), *Lecture Notes in Computer Science*, Vol. 131, Springer Verlag, 1981, pp 52–71.

[CM94] J. J. Chilenski and S. P. Miller, Applicability of modified condition/decision coverage to software testing, *Software Engineering Journal*, Vol. 9, No. 5, September 1994, pp. 193-200.

[CW+96] Edmund M. Clarke, Jeanette M. Wing, et al. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys,* Vol. 28, No. 4, December 1996, pp. 626–643.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems,* Vol. 15, No. 1, January 1993, pp. 36–72.

[DJ96] Craig A. Damon and Daniel Jackson. Efficient Search as a Means of Executing Specifications. *Proceedings Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, Passau, Germany, *Lecture Notes in Computer Science,* Vol. 1055, Springer Verlag, March 1996, pp. 70–86.

[DJJ96] Craig A. Damon, Daniel Jackson, and Somesh Jha. Checking Relational Specifications with Binary Decision Diagrams. *Proceedings 4th ACM SIGSOFT Conference on Foundations of Software Engineering,* San Francisco, CA, October 1996, pp. 70–81.

[ES94] Marcin Engel and Jens Ulrik Skakkeback. *Applying PVS to Z.* Technical Report ID/DTU ME 3/1, ProCos Project, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, 1994.

[Fag77] Ronald Fagin. The Number of Finite Relational Structures. Discrete Mathematics,

1977, Vol. 19, pp 50-58.

[IBM97] IBM Corporation. *Object Constraint Language Specification.* Version 1.1, September, 1997. Available at http://www.software.ibm.com/ad/ocl.

[JD95] Daniel Jackson and Craig A. Damon. *Semi-Executable Specifications.* Technical Report CMU-CS-95-216, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 1995.

[JD96a] Daniel Jackson and Craig A. Damon. *Nitpick: A Checker for Software Specifications (Reference Manual).* Technical Report CMU-CS-96-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[JD96b] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering,* July 1996, Vol. 22, No. 7, pp. 484–495.

[JJD96] Daniel Jackson, Somesh Jha, and Craig A. Damon. Faster Checking of Software Specifications by Eliminating Isomorphs. *Proceedings of ACM Symposium on Principles of Programming Languages (POPL '96),* St. Petersburg Beach, FL, January 1996, pp. 79–90.

[JJD98] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications. *ACM Transactions on Programming Languages and Systems,* To appear.

[Jip92] Peter Jipsen. *Computer-aided Investigations of Relation Algebras.* Doctoral thesis, Dept. of Mathematics, Vanderbilt University, Nashville, Tennessee, May 1992.

[Kum92] Vipin Kumar. Algorithms for constraint satisfaction, A survey. *AI Magazine,* Vol. 13, No. 1, Spring 1992, pp. 32–44.

[Mac92] Alan K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence,* Vol. 58, December 1992, pp. 3–20.

[McK81] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium* 21 (1981), pp 499–517.

[McK94] Brendan D. McKay. *Nauty User's Guide,* version 1.5. Computer Science Department, Australian National University, GPO Box 4, ACT 2601, Australia.

[SM96] Mark Saaltnik and Irwin Meisels. *The Z/Eves Reference Manual (draft).* Technical Report TR-96-5493-03, ORA Canada, Ottawa, Canada, December 1995, revised April 1996.

[SF91] Norman M. Sadeh and Mark S. Fox. *Variable and Value Ordering Heuristics for Hard Constraint Satisfaction Problems: An Application to Job Shop Scheduling,* CMU-RI-TR-91-23, Carnegie Mellon University, Pittsburgh, PA, 1991.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. *Proceedings 10th National Conference on Artificial Intelligence (AAI-92),* San Jose, CA, July 1992, pp. 440–446.

[Sla94] John K. Slaney. Finder: Finite Domain Enumerator, System Description. *Proceedings of 12th International Conference on Automated Deduction,* Nancy, France, Alan Bundy (ed.), *Lecture Notes in Artificial Intelligence,* Vol. 814, Springer Verlag, Berlin, 1994, pp. 798–801.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual,* Second edition, Prentice Hall, 1992.

[Wal75] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision,* ed. Patrick H. Winston, McGraw Hill, 1975, pp. 19–91.

[ZZ95] Jian Zhang and Hantao Zhang. Constraint Propagation in Model Generation. *Proceedings Principles and Practice of Constraint Programming - CP'95,* Ugo Montanari, Francesca Rossi (Eds.), Cassis, France, September 1995, Lecture Notes in Computer Science, Vol. 976, Springer 1995, pp. 398–414.

[Zha96] Jian Zhang. Constructing Finite Algebras with Falcon. *Journal of Automated Reasoning,* Vol. 17, No. 1, August 1996, pp. 1–22.